

ITU-T Software Tool Library 2000 User's Manual

ITU-T Users' Group on Software Tools

Geneva, December 2000

Copyright © 2000, 2001 by the International Telecommunication Union (ITU)

This is edition 1.0 of the “*ITU-T Software Tool Library Manual*”, for the 2000 release of the *ITU-T Software Tool Library*, distribution 3.0, February 25, 2001.

Published by the ITU. Copies of this manual are available as part of the STL2000 distribution. STL2000 copies can be acquired from:

ITU General Secretariat
Sales Service
Place du Nations
CH-1211 Geneve 20
Switzerland

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Contents

1	Introduction	1
1.1	Organization of the Software Library	2
1.2	Who to contact	3
1.3	Acknowledgements	3
2	Tutorial	5
2.1	Acronyms	5
2.2	Definition of terms	6
2.2.1	Overload point	6
2.2.2	Signal power	6
2.2.3	Signal level	7
2.2.4	Relation between overload and maximum levels	7
2.2.5	Saturation	8
2.2.6	Data representation	8
2.2.7	Data justification	8
2.2.8	Equivalent results	9
2.2.9	Little- and big-endian data ordering	9
2.3	Guidelines for software tool development	12
2.4	Software module I/O signal representation	14
2.5	Tool specifications	16
3	RATE-CHANGE: Up- and down-sampling module	19
3.1	Description of the Algorithm	19
3.1.1	High-quality	20
3.1.2	Telephony-band weighting	21
3.1.3	Wideband weighting	24
3.1.4	Noise weighting	24

3.1.5	PCM Quality	24
3.2	Implementation	25
3.2.1	FIR module	27
3.2.2	IIR Module	49
3.3	Tests and portability	59
3.4	Examples	60
3.4.1	Description of the demonstration programs	60
3.4.2	Example: Calculating frequency responses	60
4	EID: Error Insertion Device	65
4.1	Description of the Algorithm	65
4.1.1	Simple Channel Model	65
4.1.2	The Bellcore Model	67
4.2	Implementation	69
4.2.1	open_eid	72
4.2.2	open_burst_eid	72
4.2.3	reset_burst_eid	73
4.2.4	close_eid	73
4.2.5	BER_generator	73
4.2.6	FER_generator_random	74
4.2.7	FER_generator_burst	74
4.2.8	BER_insertion	75
4.2.9	FER_module	76
4.3	Tests and portability	77
4.4	Examples	77
4.4.1	Description of the demonstration programs	77
4.4.2	Using the bit error insertion routine	78
4.4.3	Using the frame erasure routine	80
5	G.711: The ITU-T 64 kbit/s log-PCM algorithm	83
5.1	Description of the algorithm	83
5.2	Implementation	85
5.2.1	alaw_compress and ulaw_compress	86
5.2.2	alaw_expand and ulaw_expand	86
5.3	Tests and portability	87

5.4	Example code	87
5.4.1	Description of the demonstration program	87
5.4.2	Simple example	87
6	G.726: The ITU-T ADPCM algorithm at 40, 32, 24, and 16 kbit/s	89
6.1	Description of the 32 kbit/s ADPCM	90
6.1.1	PCM format conversion	90
6.1.2	Difference Signal Computation	90
6.1.3	Adaptive Quantizer	91
6.1.4	Inverse Adaptive Quantizer	91
6.1.5	Quantizer Scale Factor Adaptation	91
6.1.6	Adaptation Speed Control	91
6.1.7	Adaptive Predictor and Reconstructed Signal Calculator	92
6.1.8	Tone Transition and Detector	92
6.1.9	Output PCM Format Conversion	92
6.1.10	Synchronous Coding Adjustment	92
6.1.11	Extension for linear input and output signals	93
6.2	ITU-T STL G.726 Implementation	93
6.2.1	<code>G726_encode</code>	94
6.2.2	<code>G726_decode</code>	95
6.3	Portability and compliance	96
6.4	Example code	97
6.4.1	Description of the demonstration programs	97
6.4.2	Simple example	97
7	G.727: The ITU-T embedded ADPCM algorithm at 40, 32, 24, and 16 kbit/s	99
7.1	Description of the Embedded ADPCM	99
7.1.1	Extension for linear input and output signals	99
7.2	ITU-T STL G.727 Implementation	101
7.2.1	<code>G727_reset</code>	101
7.2.2	<code>G727_encode</code>	101
7.2.3	<code>G727_decode</code>	102
7.3	Portability and compliance	103
7.4	Example code	103

7.4.1	Description of the demonstration program	103
7.4.2	Simple example	103
8	G.722: The ITU-T 64, 56, and 48 kbit/s wideband speech coding algorithm	105
8.1	Description of the 64, 56, and 48 kbit/s G.722 algorithm	105
8.1.1	Functional description of the SB-ADPCM encoder	107
8.1.2	Functional description of the SB-ADPCM decoder	109
8.2	ITU-T STL G.722 Implementation	111
8.2.1	<code>g722_encode</code>	114
8.2.2	<code>g722_decode</code>	115
8.2.3	<code>g722_reset_encoder</code>	115
8.2.4	<code>g722_reset_decoder</code>	116
8.3	Portability and compliance	116
8.4	Example code	117
8.4.1	Description of the demonstration programs	117
8.4.2	Simple example	117
9	RPE-LTP: The full-rate GSM codec	119
9.1	Description of the 13 kbit/s RPE-LTP algorithm	119
9.1.1	RPE-LTP Encoder	119
9.1.2	RPE-LTP Decoder	121
9.2	Implementation	121
9.2.1	<code>rpeltp_encode</code>	124
9.2.2	<code>rpeltp_decode</code>	124
9.2.3	<code>rpeltp_init</code>	125
9.2.4	<code>rpeltp_delete</code>	125
9.3	Portability and compliance	126
9.4	Example code	126
9.4.1	Description of the demonstration program	126
9.4.2	Simple example	126
10	Duo-MNRU: The Dual-mode Modulated Noise Reference Unit	129
10.1	Description of the Algorithm	130
10.2	Implementation	132

10.2.1	<code>MNRU_process</code>	139
10.3	Portability and compliance	140
10.4	Example code	141
10.4.1	Description of the demonstration programs	141
10.4.2	Simple example	141
11	SVP56: The Speech Voltmeter	143
11.1	Description of the Algorithm	143
11.2	Implementation	146
11.2.1	<code>init_speech_voltmeter</code>	147
11.2.2	<code>speech_voltmeter</code>	147
11.2.3	Getting state variable fields	148
11.3	Portability and compliance	148
11.4	Examples	149
11.4.1	Description of the demonstration programs	149
11.4.2	Small example	149
12	ITU-T Basic Operators	153
12.1	Overview of basic operator libraries	153
12.2	Description of the 32-bit basic operators and associated weights	153
12.2.1	Variable definitions	153
12.2.2	Arithmetic operators with complexity weight of 1	154
12.2.3	Arithmetic operations with complexity weight of 2	156
12.2.4	Arithmetic operations with complexity weight of 3	158
12.2.5	Arithmetic operations with complexity weight of 4	159
12.2.6	Arithmetic operations with complexity weight of 15	159
12.2.7	Arithmetic operations with complexity weight of 18	159
12.2.8	Arithmetic operations with complexity weight of 30	159
12.2.9	Complexity associated with data moves, logical operations, arithmetic test and other operations	160
13	UTILITIES: UGST utilities	163
13.1	Some definitions	163
13.2	Implementation	164
13.2.1	<code>scale</code>	164

13.2.2	sh2fl	164
13.2.3	sh2fl_alt	165
13.2.4	fl2sh	166
13.2.5	serialize*_justified	167
13.2.6	parallelize*_justified	168
13.3	Portability and compliance	170
13.4	Example code	170
13.4.1	Description of the demonstration programs	170
13.4.2	The master header file for the STL demonstration programs	170
13.4.3	Short and float conversion and scaling routines	171
13.4.4	Serialization and parallelization routines	173
14	References	177
A	Unsupported tools	181
A.1	Source code	181
A.2	Scripts	182
A.3	Makefiles	182
A.4	Test files	182
B	Future work	185

Chapter 1

Introduction

In July 1990, Study Group XV of the then CCITT decided to set up a group to deal with the development of common software tools to help in the development of speech coding standards. In the same period, cooperation was requested with SG XII Speech Quality Experts Group (SQEG), and a group called ‘User’s Group on Software Tools’ (UGST) was initially established with almost 20 corresponding members. The basic means of interaction was (then incipient) electronic mail (e-mail) messages, for the exchange of files and experiences. In addition to this, there were meetings held mainly during regular Working Party XV/2 (Signal Processing) sessions, where most of the decisions were made.

As result of that very intensive work, several software tools evolved forming the ‘*1992 ITU-T Software Tool Library*’ (STL92) which included, as its first application, the Qualification Test for a Speech Coder at 8 kbit/s. After this initial release, another release was approved by ITU-T Study Group 15 in May, 1996, and called STL96. The STL96 introduced substantive improvement and new features to the STL92. In November 2000, ITU-T Study Group 16 approved an updated version to the STL, the **STL2000**, which is described in this document. Terms and conditions on the usage of the ITU-T STL are found in ITU-T Recommendation G.191 [1].

The remaining chapters of this document describe the principles that guided the generation of the ITU-T STL, as well as the description of its organization. The various tools are described in separate chapters. These descriptions have the following general outline:

- a. technical description of the method or algorithm involved;
- b. description of the algorithm implementation in this release (including prototypes, parameters, returned value, etc.); and
- c. testing, applications and examples.

All the STL modules had their portability tested for MSDOS/Windows and several Unix flavors. In MSDOS, all modules were tested with the MSDOS port of the GNU `gcc` compiler (a.k.a. DJCPP) and with at least one of these Borland compilers: Turbo C 2.0, Turbo C++ 2.0, or Borland C++ 3.1. In the Windows environment, the code was tested using MS Visual C version 6.1 SP3 as well as using the `gcc` compiler available in the Cygnus CYGWIN development environment (www.cygnus.com). The VAX/VMS environment was fully supported in the STL96 (VAXC and `gcc`), however it was not possible to continue it for the STL2000 due to operational reasons. However, compilation

under gcc should provide the expected results. It should be noted however that some tools were tested for Ultrix. For the Unix operating system, portability was verified for three workstation platforms: Sun Solaris 5.7 (SPARC or Intel CPUs, using gcc), HP 9000 Series 700 HP-UX 9.05 or 10.20 (using gcc), and Silicon Graphics. On Silicon Graphics systems, the standard cc compiler was used.

1.1 Organization of the Software Library

Each tool of the STL has been produced as a stand-alone module, such that it may be linked to a user's program, application or system. In the present version, there are several of these modules:

1. **RATE-CHANGE:** An up- and down-sampling algorithm with embedded filterings:
 - ITU-T Rec. G.712 filter for factors of 1:2, 2:1 and 1:1
 - High-quality filter for factors 1:2, 2:1, 1:3, and 3:1
 - IRS send-side weighting filter, for several sampling rates: 8, 16, and 48 kHz. This includes the "full-IRS" as in ITU-T Rec. P.48 as well as the "modified" IRS as in Annex D of ITU-T Rec. P.830.
 - Modified-IRS receive-side filter is also available for 8 and 16 kHz sampled data.
 - Δ_{SM} weighting filter for near-to-far field conversion
 - Psophometric weighting filter of ITU-T Rec. O.41 for noise measurements
 - ITU-T P.341 weighting filter for wideband signal (50-7000 Hz)
2. **EID:** Error insertion algorithm, with routines for generation of bit error patterns (random or burst) as well as random and burst frame erasure.
3. **G.711:** The 64 kbit/s PCM algorithm with A and μ law of ITU-T Rec. G.711.
4. **G.726:** The 40, 32, 24, and 16 kbit/s ADPCM algorithm of ITU-T Rec. G.726.
5. **G.727:** The 40, 32, 24, and 16 kbit/s embedded ADPCM algorithm of ITU-T Rec. G.727.
6. **G.722:** The 64, 56, and 48 kbit/s wideband speech ADPCM algorithm of ITU-T Rec. G.722.
7. **RPE-LTP:** The 13 kbit/s RPE-LTP algorithm of the full-rate GSM system (GSM Rec. 06.10).
8. **MNRU:** The modulated noise reference unit of ITU-T Rec. P.810 (formerly ITU-T Rec. P.81).
9. **SVP56:** The Speech Voltmeter for measuring the active speech level (which skips over silence in a utterance) of ITU-T Rec. P.56.

10. **BASOP:** The set of basic digital signal processing (DSP) operators that represent the set of instructions typically available in digital signal processors.
11. **UTILITIES:** Tools that have been developed to assure proper interfacing between the various tools. These tools do not relate to any ITU-T Recommendation. Included are tools for conversion between float and short data representations, between parallel and serial (bit-stream) formats, and for scaling of data.

It should be noted that C code is available for a number of codecs as a normative part of the respective standards, e.g. ITU-T G.723.1, G.729, G.722.1; ETSI GSM-HR, GSM-EFR, GSM-AMR; TIA IS-641, IS-127, IS-96A, among others. These source codes are not appropriate for inclusion in the ITU-T STL for a number of reasons: they are an integral part of the respective standards, are maintained within the scope of the respective standards development organizations (SDOs), are protected by copyrights, and are openly available. Parties interested in acquiring these source codes should contact the appropriate SDO.

1.2 Who to contact

In case of problems with any of the tools, the UGST coordinator is identified as the contact point. In addition to the description of the problem, it is recommended that the user's address should be sent to the contact point, so that the user can be kept informed of changes and new releases of the ITU-T Software Tool Library. The address is:

Simão Ferraz de Campos Neto
Lockheed Martin Global Telecommunications (LMGT)
22300 COMSAT Drive
Clarksburg, MD 20871
United States
Fax: +1-301-428-9287
E-mail: simao.campos@labs.comsat.com

1.3 Acknowledgements

Several organizations which participate in ITU-T Study Groups 12, 15 and 16 have substantially contributed to the completion of this release of the ITU-T STL.

First and foremost, UGST wishes to thank CPqD/Telebrás (Brazil) for its support of the early coordination (1990-1993) of the activity and of the development of the following tools: Utilities, G.711, G.726, MNRU, and SVP56. For the first two, the work was shared with PKI (Germany), which also provided the initial version of the modules EID and RATE-CHANGE, as well as basic material that supported the initial organization of the work, together with Telenor (formerly NTA, Norway) and the DBP-Telekom (Germany). DBP-Telekom also collaborated in providing several software tools used in the Host Laboratory for the ITU-T 8 kbit/s speech coder: modified IRS filters, adaptation

of the Bellcore burst frame erasure model, and Δ_{SM} filter. UGST also wants to thank CSELT (Italy) for making available its Fortran MNRU program, which was the starting point of the present implementation, and for the implementation of the psophometric filter. CNET (France) provided the G.722 tool, which was greatly appreciated. UGST kindly thanks Mr. Jutta Deneger for allowing the incorporation of his implementation of the RPE-LTP algorithm in the STL. Also, Bellcore provided several programs in Fortran and C that, while not used directly in the present version of the STL, were important in various stages of the development of the Library, especially a version of the Red Book G.721. PTT Ukraine graciously provided the G.727 implementation, which was warmly welcomed. COMSAT Labs (now part of Lockheed Martin Global Telecommunications, LMGT), in turn, provided essential help in funding the recent coordination work (1994-current), and the harmonization and documentation of the tools. Also important was the testing work done by the Research Institute of the Deutsches Telekom (now T-Nova/DT), as well as PKI, Telebrás, AT&T (USA), and CNET.

Several parts of this manual were possible only by the contribution of several individuals: Pierre Combescure (CNET) for the description of the G.722 algorithm, Rudolf Hofmann (PKI), for description the Gilbert-Elliot channel implemented in the EID module, Peter Kroon (AT&T) for the description of the RPE-LTP algorithm, and Vijay Varma (Bellcore) for the text describing the Bellcore Burst Error Model. It is also necessary to thank all UGST members that collaborated to make this version of the STL Manual more accurate.

Chapter 2

Tutorial

2.1 Acronyms

Several acronyms are used in this text. The most important are:

ANSI ... American National Standards Institute.

BBER .. Burst Bit Error Rate

BER Bit Error Rate (refers to *random* bit errors)

BFER .. Burst Frame Erasure Rate

DAT Digital Audio Tape.

EID Error insertion device.

ETSI ... European Telecommunications Standards Institute.

FER Frame Erasure Rate (refers to *random* frame erasures)

GSM ... Global System for Mobile Communications. Pan-European digital-cellular system operating at a net rate of 13 kbit/s in its full-rate system.

IRS Intermediate Reference System, defined in ITU-T Rec. P.48 for the so-called “full-IRS” mask, or in Annex D of ITU-T Rec. P.830 for the so-called “modified” IRS mask.

ITU International Telecommunication Union.

ITU-T .. Standardisation Sector of the International Telecommunication Union.

LSb Least significant bit.

MIRS ... Modified-IRS telephony speech weighting (in ITU-T Rec. P.830 Annex D).

MSb Most significant bit.

PSTN .. Public Switched Telecommunication Network.

R&O ... Requirements and Objectives, for performance of software tools.

SQEG .. Speech Quality Experts Group, of Study Group 12 of the ITU-T.

STL92 .. ITU-T Software Tools Library, release 1992.

STL96 .. ITU-T Software Tools Library, release 1996.

STL2000 ITU-T Software Tools Library, release 2000.

UGST .. Users’ Group on Software Tools, of Study Group 16 of the ITU-T.

2.2 Definition of terms

In the documentation of the ITU-T software tools, several terms are widely used and are defined below.

2.2.1 Overload point

The overload point within the digital domain is defined by the (normalized) amplitude value.

$$x_{\text{over}} \triangleq 1.0$$

How this overload point relates to the analogue world depends on the conversion method between the analog and digital domains, and is beyond the scope of this document. All signals in this manual are relative to this overload point in the digital domain.

NOTE: This overload point does NOT depend on the quantisation method used and remains identical, regardless of whether the quantisation is done e.g. with 32, 16, 13 or 8 bits.

1. In floating point (either single or double precision), the representation of this value is exact. In this text, and also in the tools, this data type is called **float**.
2. In 32 bit 2's complement representation the data can be represented by multiplying the normalized value by 2^{31} . For example, the largest possible positive value is represented by **0x7FFFFFFF**. The largest negative value is represented by **0x80000000**. In this text, and also in the tools, this data type is called **long**.
3. In 16 bit 2's complement representation the data can be represented by multiplying the normalized value by 2^{15} . For example, the largest possible positive value is represented by **0x7FFF**. The largest negative value is represented by **0x8000**. In this text, and also in the tools, this data type is called **short**.
4. The statements above may be generalized for all wordlengths in fixed point representation. The idea is to set the decimal point just after the MSb (sign bit).

2.2.2 Signal power

The power of a signal $x(n)$ with a length of N samples is defined by

$$P = \frac{1}{N} \sum_{n=0}^{N-1} x(n)^2$$

A signal which does not contain amplitude values exceeding the overload point can have a maximum signal power of 1.0. This is the power of a DC signal with an amplitude of 1.0 or of any other signal comprising only the values ± 1.0 (e.g., a square wave signal).

2.2.3 Signal level

The power level in decibels is defined relative to a reference power level $P_0 = 1.0$:

$$L = 10 \log_{10}(P/P_0) \text{ (dBov)}$$

The level of a signal power $P = 1.0$ is thus 0 dBov (where the characters “ov” arbitrarily mean digital **o**verload signal level), which is chosen to be the reference level. A signal with such power level could be either (a) a sequence of maximum positive numbers (+1), (b) a sequence of maximum negative numbers (-1), or (c) a rectangular function exercising only the positive or negative maximum numbers (± 1). The level of a sinewave with an amplitude (peak value) of 1.0 is therefore $L = -3.01$ dBov.

2.2.4 Relation between overload and maximum levels

The measurement of signal levels in the digital part of the network is normally expressed by telecommunications engineers as y dBm0, i.e., the level relative to 1 mW in 600Ω . However, from the software point of view, it is more convenient to represent levels relative to the maximum power that can be stored in integer format on a computer, e.g. z dBov. A conversion between both representations can be expressed as:

$$y \text{ (dBm0)} = z \text{ (dBov)} + C$$

For the G.711 encoding rule, a sinewave which exercises the maximum level has a power T_{max} of 3.14 dBm0 for A-law, and of 3.17 dBm0 for μ -law. On the other hand, the RMS level of these sinewaves would always be -3.01 dBov. Therefore, C above becomes 6.15 dB for A-law and 6.18 dB for μ -law. For the G.722 wideband coding algorithm, the overload point of the A/D and D/A converters should be 9 dBm0. Therefore, in that case, C becomes 12.01 dB.

The following relationships summarize the discussion:

$$\Lambda_A(\text{dBm0}) = L_{ov}(\text{dBov}) + 6.15\text{dB(A-law)}$$

$$\Lambda_\mu(\text{dBm0}) = L_{ov}(\text{dBov}) + 6.18\text{dB}(\mu\text{-law})$$

$$\Lambda_{wb}(\text{dBm0}) = L_{ov}(\text{dBov}) + 12.01\text{dB(G.722)}$$

2.2.5 Saturation

Saturation is the limitation of signal amplitudes to values equal to or smaller than the overload point:

$$y(k) = \begin{cases} -1.0, & \text{if } x(k) < -1.0 \\ x(k), & \text{if } -1.0 \leq x(k) \leq +1.0 \\ +1.0, & \text{if } x(k) > +1.0 \end{cases}$$

2.2.6 Data representation

Unless otherwise noted all waveforms within the signal processing are assumed to have infinite precision and unlimited amplitude. The overload point is therefore the reference point only. In practice these signals may well be represented in 32 bit floating point arithmetic or high precision integer arithmetic (24 bit for data and coefficients, 48 to 56 bit for products and accumulation). In most cases, 16 or 32 bit integer arithmetic is not precise enough.

Signals derived from 16 bit 2's complement representation (DAT, files, digital I/O interface) should be converted to this (approximately) infinite precision before processing by modules that require floating point input. Normalization of the floating point values to the overload point is recommended.

2.2.7 Data justification

Justification of data here is used without distinction to data alignment and data adjustment: where the upper or lower significant bit of an integer sample is located.

Left-justified data are samples whose most significant bit is located at the leftmost position of the computer storage unit used for it. Remaining low-bit positions must be set to zero.

Right-justified data are samples whose least significant bit is located at the rightmost position of the computer storage unit used for it. Remaining upper bits depend on the data representation: if two's complement, sign extension from sample's MSb to storage's MSb is needed; otherwise, the upper (unused) bits shall be zeroes.

As an example, suppose a 12-bit resolution, two's complement sample, to be stored for processing in a **short**. If left-justified, then a sign bit (the MSb!) is found in bit 15 (the MSb) of the **short** that stores it. On the other hand, if right-justified, the LSb will be the bit 0 of the **short**, in this case. If it is a negative number, there would be sign extension for bit 12 to 15. If it is an unsigned number, the upper 4 bits (in the example) are all zeros. Figure 2.1 illustrates these three cases.

<i>Bit number</i>	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>Bit type</i>	s	v	v	v	v	v	v	v	v	v	v	v	x	x	x	x

(a) Left-justified data

<i>Bit number</i>	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>Bit type</i>	s	s	s	s	v	v	v	v	v	v	v	v	v	v	v	v

(b) Right-justified, sign-extended data

<i>Bit number</i>	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>Bit type</i>	0	0	0	0	v	v	v	v	v	v	v	v	v	v	v	v

(c) Right-justified, unsigned data

Figure 2.1: Illustration of a left- and right-justified data with 12-bit resolution. Bit types *s*, *v*, and *x* represent respectively sign bit(s), valid bits and unused bits.

2.2.8 Equivalent results

Several software tools, such as the G.711 algorithm, are defined in terms of precise fixed point operations. Therefore, when comparing the output of one of these algorithms on different platforms, or for compilation using different C compilers, one should expect identical sample values for reference processed materials.

Other algorithms, however, may include highly intensive processing, or complex mathematical functions. Examples of these are rate change filters and floating-point arithmetic speech coders, such as the 16 kbit/s LD-CELP of ITU-T Rec. G.728. In such cases, it is expected that the processing of the same reference material on different platforms will generate almost identical results. The generated files will probably be identical for most of the samples, and for some samples they will differ by a small amount, e.g. ± 1 , or more rarely by ± 2 or more. For the purposes of the STL, such an implementation is said to produce *equivalent results* on different platforms.

2.2.9 Little- and big-endian data ordering

Present computer systems agree only on the data access for byte-oriented data structures. Although computer systems exist whose bytes do not have 8-bits, the majority of the systems implement bytes as 8-bit data structures. In general, the computer architectures do not differ in the way they access the bit-order within a byte. In other words, for the vast majority of the computer systems existing today, the least significant bit occupies the lower memory position (i.e., bit 0), and the most significant bit occupies the higher memory position in the byte (i.e., bit 7). In terms of C operations, if **b** is a byte structure, then **b&0x1** returns the LSb, and **(b>>7)&0x1** returns the MSb.

Although most computer architectures agree on the definition of a byte and how its bits

Table 2.1: Example of big- and little-endian systems

Big-endian		Little-endian	
Computer	Microprocessor	Computer	Microprocessor
Sun-3	Motorola 68000 family	IBM-PC/compatibles ^(a)	Intel 80x86/Pentium
Sun-4	Sun SPARC family	DEC-Stations	MIPS RISC
Silicon Graphics	MIPS RISC	DEC Alpha	DEC Alpha AXP
IBM 370	IBM	VAX/VMS Microcomputers	VAX CPU
HP 9000-700	HPPA RISC		
Legend:	CISC: Complex Instruction Set Computer		
	RISC: Reduced Instruction Set Computer		
Note:	(a) Including Windows 9x/NT/2000 and Linux and Solaris on Intel CPUs.		

are accessed, they vastly differ on how multi-byte structures are accessed. Trivial examples of multi-byte structures are 16-bit **short** words or 32-bit **long** words. There are currently two access means currently implemented by different CPUs in the market, which differ on the significance of the bytes that are first read from memory positions.

On the so-called *big-endian* systems, the first byte read from a multi-byte structure is always the most significant byte. For example, if the two bytes 0x12 (low address) and 0x34 (high address) are stored in two consecutive memory addresses, then the number read and stored in the CPU accumulator would be 0x3412, or 13330 in decimal. The big-endian data organization is, for this reason, also known as *high-byte first*.

For the so-called *little-endian* systems, the first byte read from a multi-byte structure is always the least significant byte. For this reason, the little-endian data organization is also known as *low-byte first*. Using the same example as before, for the two consecutive bytes in memory 0x12 and 0x34, the value loaded on a little-endian CPU will be 0x1234, or 4660 in decimal.

The concept is extended to other multi-byte data structures, such as 32-bit or 64-bit integers. For example, the consecutive bytes 0x12, 0x34, 0x56, and 0x78 would be loaded as the 32-bit integer 0x78563412 on the accumulator of a big-endian CPU and as 0x12345678 on the accumulator of a little-endian CPU.

Table 2.1 indicates the data organization for several computer platforms. It should be noted that the data organization is a function of the CPU family rather than of the operating system used. For example, Solaris on Sparc platforms uses big-endian data organization, while Solaris on Intel 80x86/Pentium platforms uses little-endian data organization. Similarly, most Linux systems are little-endian (because they run on Intel 80x86/Pentium CPUs), but several other implementations are actually big-endian (e.g. PowerPC CPU used in Macintosh machines).

The segment of C code in figure 2.2 can be used to determine whether a given computer system has big- or little-endian data organization.

The approach above determines whether a platform is big- or little-endian, but it does not answer the question of what is the byte orientation in a given file. Although there is no closed-form method for such a determination, there is an empirical method that can be carefully used for speech signals (usually represented using 16-bit linear PCM words)

```

#include <stdio.h>
#include <string.h>

int is_little_endian()
{
    /* Hex version of the string ABCD */
    unsigned long tmp = 0x41424344;

    /* Compare the hex version of the four characters with the ASCII version */
    /* On big-endian (or high-byte-first) systems, 0x41 ('A' in ASCII) */
    /* is stored in the first memory position, and the equivalent string */
    /* is "ABCD". On a little-endian (or low-byte-first) system, 0x41 is */
    /* stored in the last position, and the equivalent string will be */
    /* "DCBA". Function strncmp will return 0 if both strings are equal */
    /* upto the first four characters. */
    return(strncmp("ABCD", (char *)&tmp, 4));
}

void main()
{
    printf("System is %s-endian\n", is_little_endian()? "little" : "big");
}

```

Figure 2.2: Sample code for determination of byte organization.

based on two speech properties: speech signals follow a gamma distribution (hence most of the samples have small amplitude), and levels in voiced segments are usually in the -15 dBov through -40 dBov range. For files that have a byte orientation mismatching that of the computer platform, the mostly small samples of the speech signal will be measured as having large amplitude. Hence, if a high-level power is found when measuring the power of a voiced segment (typically around -4 dBov), one can assume that the file needs to be byte-swapped. It is important however to measure the level for *voiced segments*, since for silent intervals the increase in gain is not so dramatic and will not allow for a conclusion on the byte-orientation of the file.

When the change of format is necessary for **short** and **long** data, the operations in figure 2.3 should be used. The conversion between big- and little-endian data representation for 16-bit data is simple and is known as *byte swapping*. The byte swapping operation can be implemented in several fashions. For example,

```

short swap_one_short(short in)
{
    return (((in>>8)&0xFF) | (in<<8));
}

```

It should be noted that the simple byte-swapping above does not work properly for conversion of other multi-byte structures. For the purposes of the STL, however, 16-bit structures is the most import case. For several of the STL modules, the provided test files

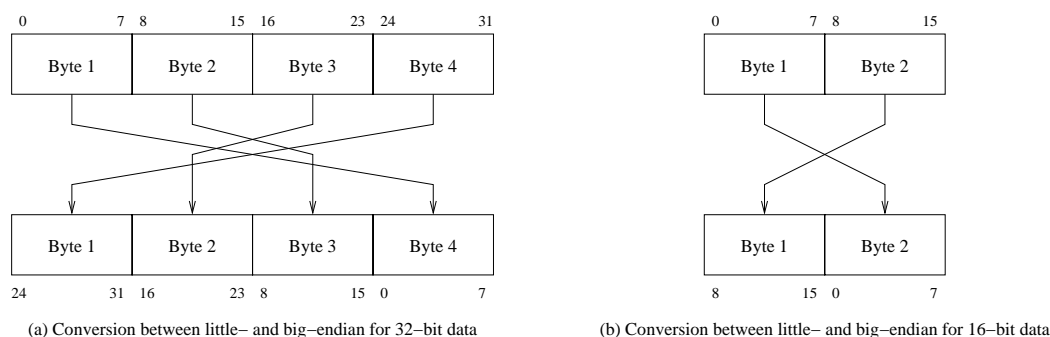


Figure 2.3: Conversion between big- and little-endian

in general need to be byte swapped in one or another computer platform. The documentation and the “manifesto” accompanying each software tool module describe which files, if any, should be byte-swapped on certain platforms. As default, binary files organized in 16-bit words are provided in big endian format in the STL distribution.

2.3 Guidelines for software tool development

The software tools provided by the ITU-T User’s Group on Software Tools are to be used by laboratories with different computers and A/D-D/A equipment. To make the software accessible to everybody, it should be highly portable across operating systems and allow for easy implementation in existing hardware environments.

To achieve this, some simple guidelines were followed in the development of the tools. The following are the UGST guidelines used to generate the official and beta releases of the ITU-T Software Tool Library.

- i. All software should be written in ANSI C.
- ii. Features of the language whose representation may create side-effects should not be used (e.g. `union`).
- iii. All variables must be declared and the types used in the declarations must be the least platform dependent. For example, the keyword `int` must be avoided. Instead `short` should be used for 16-bit integers and `long` should be used for 32-bit integers.
- iv. The software should not contain any input or output that may be system dependent (e.g. open, read and write file operations). Instead, data must be passed to the modules as parameters of function calls. This will allow each laboratory to integrate the modules with their own application software without changing the modules. Interfaces to various file formats and user interaction can optionally be provided as example main programs¹ that will not be a part of the library module and should contain the least possible amount of code.

¹Also called “*demonstration programs*” in this manual.

- v. Well defined digital signal formats should be used and documented for each module to allow the various modules to work together.
- vi. The interface to the file system should be made in a standard way, but only within the example programs.
- vii. The source code should be properly documented, with a standard header.
- viii. Modularity is encouraged in the software design. All modules are self-contained, i.e. global definitions should be avoided.
- ix. Each module should have an attached specification document explaining the function and use of the module, the level of detail depending on its complexity.
- x. The software modules shall be distributed to interested laboratories for comments and testing before they are approved and included in the ITU-T Software Tools Library, to minimize the occurrence of bugs and to assure conformance with related ITU-T Recommendations (when applicable). Two test procedures have been devised: compliance and portability.

The *compliance procedure* (or compliance test) is to certify that a given tool module fully complies with specifications, which should be carried out by at least one organization other than the proponent organization (or by a group of organizations, each one checking a different subset of the specifications, such that all together cover all the specifications). In order to minimize the probability of systematic errors, these procedures should be defined by the verifying organization(s) without input from the tool provider(s).

The *portability verification procedure* (or portability test) is to certify that a given validated tool works on platforms other than the one(s) where they were generated and validated. In simple cases these verification procedures could be just test vectors (e.g. speech or noise files). It was also pointed out that problems may arise in Unix platforms, due to the existence of several flavors of Unix available today (this means that a verification procedure could be valid in one Unix machine, but not in other).

Portability verification procedures should be provided by the proponents and shall be run on at least two relevant operating systems (DOS, UNIX). In the past, procedures for the VMS operating system used to be required, however this operating system has become less common. For DOS, the “pure” 16-bit mode has become less common, and 16-bit emulation window under a 32-bit version of MS Windows is now prevalent. These facts affect the choice of compiler.

The following is a list of compilers used to test the portability of tools in the STL, although not all tools were necessarily tested with all compilers.

- | | |
|---------------|--|
| <i>HP/c89</i> | This is the c89 compiler that can be purchased from HP for use in HP-UX systems. For the STL, tests with this compiler were performed with HP-UX 9.05. |
| <i>HP/gcc</i> | This is the HP-UX port of the gcc compiler. The specific version may differ from tool to tool. Versions used included gcc 2.7.2.2 for HP-UX 9.05 and gcc-2.95.2 for HP-UX 10.20. |

<i>MSDOS/gcc</i>	This is the MSDOS-6.22 port of the <code>gcc</code> compiler version 2.6.3-DJGPP V1. This is a 32-bit compilation of the code, however using a 16-bit interface. Executables are not likely to run under Windows MS-DOS emulation window. Needs a run-time 32-bit extender called <code>go32.exe</code> .
<i>MSDOS/tcc</i>	This is the Borland Turbo C++ Version 1.00 <code>tcc</code> compiler.
<i>MSDOS/bcc</i>	This is Borland C++ <code>bcc</code> compiler. Versions used included 3.0 and 4.5.
<i>Solaris/gcc</i>	This is the <code>gcc</code> compiler version 2.95 running under Solaris 7, usually in a Sparc platform.
<i>SunOS/cc</i>	This is the basic <code>cc</code> C compiler bundled in the SunOS distribution. For the STL, SunOS version 4.1.3 was used.
<i>SunOS/acc</i>	This is the licensed <code>acc</code> C compiler sold by Sun Microsystems. For the STL, SunOS version 4.1 was used.
<i>Win32/gcc</i>	This is the <code>gcc</code> compiler version 2.95 running under Windows NT 4 SP 4 and with the CYGWIN Unix emulation interface. These executables need either the CYGWIN environment or the run-time library <code>cygwin1.dll</code> to run, and they are expected to work properly in a DOS emulation window under Windows 95/98 as well. This version will not run under native MS-DOS.
<i>Win32/cl</i>	This is the command-line <code>cl</code> version 12.00.8168 C compiler of the MS Visual C V.6 SP3 running under the WinNT 4 SP4 (the executables will also run in Windows 95/98/SE/Me/2000). This version will not run under native MS-DOS.

2.4 Software module I/O signal representation

The idea behind the choice of the convention in this section is that all software modules within the ITU-T tool library should be independent building blocks which can easily be combined by connecting the output of one module to the input of the next module. With this characteristic, various systems may be very easily constructed. The individual software modules must have well-defined interfaces to allow such simple connections, especially at the I/O level. This convention is based on the following:

1. All modules work ‘from RAM to RAM’. This means that the working modules are independent from physical I/O functions which are normally machine dependent. This approach also allows easy cascading of modules within one ‘main’ program.
2. All signals at the I/O interfaces of modules are represented in one of the following ways:

- (a) in single or double precision (32 or 64 bit) floating point representation. The normalized signal is used directly (*overload point = reference point = 1.0*)
 - (b) in 32 bit 2's complement representation. The normalized signal must be multiplied by 2^{31} (i.e. the decimal point is just after the MSb, same as for 16 bit representation). If less than 32 bits are required, then the signal is left adjusted within the 32 bit longword and the LSbs are optionally set to 0.
 - (c) in 16 bit 2's complement representation, as described in section 2.2.1. If less than 16 bits are required, then the signal is left adjusted (left-justified) within the 16 bit words and the LSbs are optionally set to 0. If the host machine does not provide a format with 16 bit width, then the next longer wordlength should be used with the 16 bits right adjusted.
3. Data exchange with a module shall be done directly within the calling statement (not by global variables).
 4. Data exchange with a module shall be done sample-by-sample (FIR-filtering, MNRU, etc.) or frame-by-frame (block oriented speech codec, etc.), whichever is more convenient. Larger blocks may be formed (e.g. 128 samples at a time) for better efficiency, however the block size should be rather small (less than 512). The block and its length shall be variables.
 5. All modules shall be constructed in a way that infinitely long signals may be processed with a reasonable amount of internal storage. As an example, the 'main' program could read a block of input data (e.g., next frame of time signal samples) from the disk, call a module or sequence of modules, write the output signal (e.g., next frame of coded parameters) back onto disk. This process is repeated for all the input data blocks of interest.
 6. All modules shall have
 - (a) an initialization part (if necessary) and
 - (b) a working part

The initialization part may be necessary to reset internal state variables, define the mode of operation (e.g. MNRU-mode), and so on. It is called only once at the beginning or whenever a reset to an initial state is needed.

NOTE: All state variables (if any) must be initialized at execution time, not at compile or load time.

The working part performs the processing itself. It leaves all state variables in a well-defined manner for the immediate use within the next call. One possible way to do this is to introduce a flag-variable within the call statement (e.g., named 'Initialize') which is set by the 'main' program to '1' for initialization and is set to '0' during normal operation. In this way, only one function for one module is necessary. Alternatively, a specialized initialization routine may be written, to be called before the main processing routine of the module. Only one of the approaches will be followed in the future. However, both are present in the current version of the STL.

7. The RAM allocation shall in principle be split into 'static' and 'temporary' parts. 'Static' means that the contents must be saved from call to call, preferably by means of state variables rather than truly static variables². 'Temporary' means that the contents are not saved between successive calls of the module.
8. All modules are separated in clearly and independently defined functions, but accompanied by an example 'main' program which may also include file I/O.

2.5 Tool specifications

For each tool, there are 'Requirements and Objectives' (R&Os) associated. Each of the R&Os has both a general and a specific part.

The general part includes the following³:

1. Portability among platforms and Operating Systems (DOS, UNIX, and VMS):
 - compilation [GL-i];
 - usage of language features that may cause side-effects [GL-ii];
 - usage of language features that may be ambiguous among platforms [GL-iii];
 - usage of system dependent calls (to access resources such as files, etc. within the modules) [GL-iv];
2. Efficiency:
 - use of CPU (i.e., execution speed);
 - use of I/O (intensity of access to files, etc.);
 - use of memory (physical/virtual);
 - code's coverage (verbosity versus laconism);
3. Documentation:
 - Self-documentation (e.g., comments, variables and structure resembling ITU-T Recommendations, etc.) [GL-vii];
 - Separate documentation (clarity, objectivity, etc.) [GL-ix];
4. Modularity [GL-viii]
5. Fixed point versus floating point implementations;

Following are descriptions of each of the General R&Os. Full description of the R&Os can be found in [2, Annex 4].

General performance specification refers to the document that specifies the tool in question, e.g. an ITU-T recommendation or ANSI or ETSI standard.

²As a rule, state variables should not be defined as truly static ones because this may cause side-effects.

³GL*x* refers to the Guideline number *x* in section 2.3, e.g., GLiii is the Guideline iii.

Portability addresses several points related to the tool's capacity of working on several platforms: *Compilation and linkage* refers to the necessity of changes in the source code to make a tool compile without any modification in a given environment. It was identified that the operating systems of most interest are DOS and Unix (both BSD and System V). *Side-effectable features* are those that, if used in a program, when changing one parameter, may cause other(s) to be changed implicitly. *Ambiguous features* are those that, due to the flexibility left in the C language specification, are implemented in different ways for different platforms. For example, `int` in C is 32-bit wide in VAX-C and Unix workstations, but is 16-bit wide for most compilers available on MS-DOS (Turbo-C/MS-C). *System-dependent calls* are calls that are restricted to or are implementation of features of a particular platform, to make better use of that particular computer architecture.

Efficiency is related to how the computer's resources are used in terms of CPU, I/O and memory allocation, that may be a burden and prevent the usage in some systems, either by lack of resources or length of time needed for execution. Efficiency also includes *code's coverage*, expressing how frequently code is accessed.

Documentation refers to how to describe the tool. *Self-documentation* is the documentation present in the program itself to assure that the code clearly describes the algorithm implemented, to provide compilation and linkage instructions, as well as to report known bugs, etc. A *separate document* will be mandatory when no written description of the algorithm is available, or when the written documents that specify the tool are too general.

Modularity degree is the degree of isolation that a particular tool has. From UGST Guidelines, all tools must be modular, i.e., self-contained blocks; nonetheless, tools may make use of system resources other than memory and CPU.

Arithmetic is the number representation specification, whether fixed (2's complement, 1's complement, etc.) or floating point. Here, "fixed point" shall always be understood as 2's complement representation, except where otherwise noted.

Chapter 3

RATE-CHANGE: Up- and down-sampling module

In certain applications involving digitized speech, such as subjective evaluation of speech processed by digital algorithms, it may be preferable to use sampling higher than the typical rate used for the algorithms under test. This is desirable because simpler analog filters with less phase distortion can be built. Another advantage is that upper frequency components of the signal are not lost. It also allows for the convenient shaping of the input signal, such as IRS, Δ_{SM} , and psophometric weightings. Consequently there is a need to adapt the sampling rate of the digitized signal to that of the processing algorithm. For telephony applications, the typical sampling rate is 8000 Hz with a signal bandwidth in general of 300–3400 Hz, and for wideband speech applications, a bandwidth of 50-7000 Hz is desired with sampling rate of 16000 Hz. Therefore, sampling rates above 8000 Hz and 16000 Hz are desirable, respectively. In several recent experiments [3] the sampling rate was 16 kHz. In another (see [4]), 48 kHz was utilized. Hence the need for a software tool to carry out filtering and sampling rate change. Next, the rate change and spectral weighting routines implemented in the ITU-T STL are presented.

3.1 Description of the Algorithm

Signal processing theory describes the basic arrangement for decimation of signals; first the signal is low-pass filtered to limit its bandwidth in order to avoid aliasing when the rate is lowered and, second, to decimate the samples, i.e., to drop out samples from the input signal, such that the desired output rate is obtained. For example, if a rate reduction from 48 kHz to 8 kHz is desired, a decimation factor of 6:1 is necessary. This is equivalent to say that, after limiting the bandwidth of the digitized speech to 4 kHz, 5 out of 6 samples are skipped, or alternatively, only 1 out of 6 samples will be kept (or saved) from the signal.

The up-sampling of signals requires that each of the input samples be followed by a number of zero samples, such that the desired output rate is achieved; after this, an interpolation operation of these zero samples is performed to obtain a continuous-envelope signal. For example, up-sampling data from 8 kHz to 16 kHz requires interleaving each sample of the

input signal with a zero sample followed by interpolation of the signal. This interpolation can be carried out by means of a polynomial, which is equivalent to a filtering operation.

The type of filtering required is determined by the application intended for the signals. For the tools needed in this version of the STL, three different groups of characteristics were defined:

- **High-quality:** Change in rate without changing the frequency response of the input signal. This is accomplished with a flat, linear phase, low-pass or bandpass FIR filter.
- **Spectral weighting:** Spectral weighting without rate change is necessary for some applications. For narrow-band speech, available are the IRS weighting specified in ITU-T Rec. P.48, the so-called “modified” IRS (annex D of ITU-T Rec. P.830), the far-to-near-field conversion Δ_{SM} weighting, and the psophometric noise weighting of ITU-T Rec. O.41. For wideband signals, the mask for wideband handsets, as defined in ITU-T Rec. P.341, is also available.
- **PCM quality:** Change in rate accompanied with modification of the frequency response of the input signal according to the mask specified in ITU-T Recommendation G.712. This is accomplished with a non-linear phase low-pass IIR filter.

3.1.1 High-quality

The response of the filters in this type of rate change must minimize phase and amplitude distortion. For example, for decimation from 48 kHz to 16 kHz, the filter must be flat up to about 8 kHz (except for the transition, or cut-off, region), with a linear phase. In other cases, it may be desirable to remove the DC component and hum noise (50–60 Hz AC line noise) from the signal without additional phase distortion to the upper region of the spectrum.

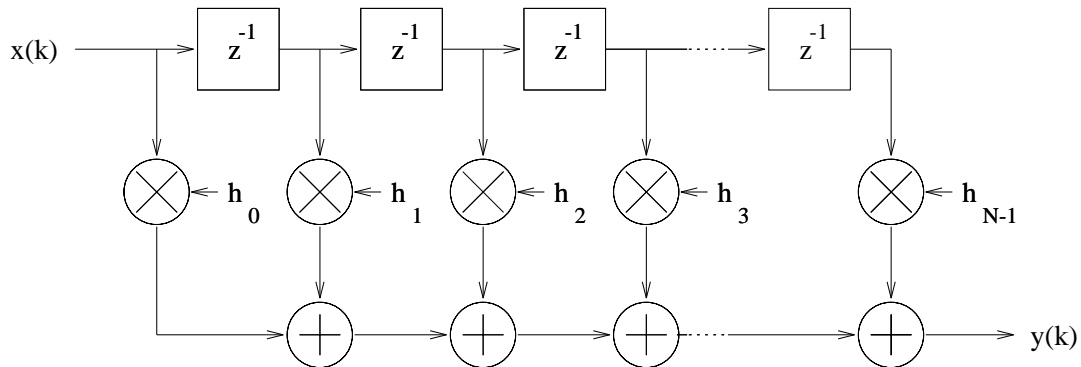


Figure 3.1: FIR filter block diagram.

One way to do this is to use a linear phase finite impulse response (FIR) digital filter, as in figure 3.1. The input and output characteristic is defined by:

$$y(k) = \sum_{i=0}^{N-1} h(i) \cdot x(k-i)$$

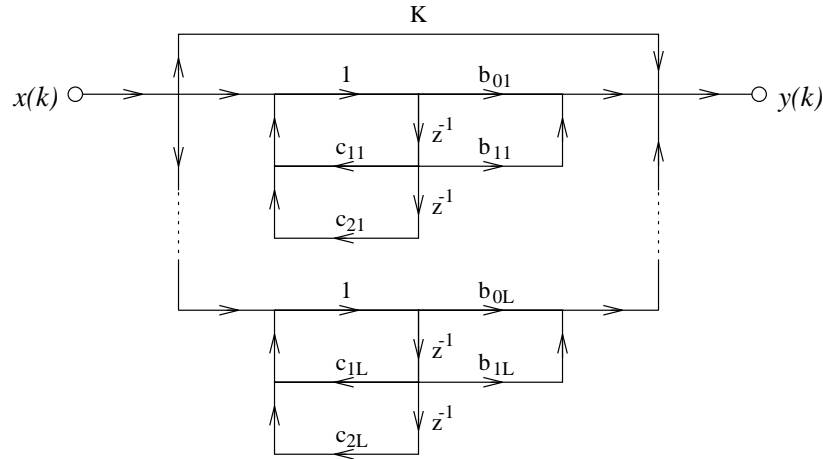


Figure 3.2: Parallel-form IIR filter block diagram.

Linear phase is guaranteed if the filter is symmetric, i.e.:

$$h(k) = h(N - 1 - k), \text{ for } k = 0..N - 1$$

3.1.2 Telephony-band weighting

IRS weighting

The IRS weighting corresponds to a bandpass filtering characteristic whose mask can be found in ITU-T Recommendation P.48 [5]. The send and receive spectral shapes of the IRS weighting were obtained in a round-robin series of measurements made on a number of contemporary analog telephones in the early 1970's [6]. From these measurements, the average send and receive frequency-response characteristics were derived. However, for the loudness balance purposes for which the IRS was designed, it was also necessary to include a 300-3400 Hz bandpass filter, known as the SRAEN filter. The values of send and receive sensitivity currently given in ITU-T Rec. P.48 (columns 2 and 3 in Table 3.1) are therefore composed of the average send and receive responses for a number of telephones, as well as the response of the SRAEN (*Système de Référence pour la détermination de l'Affaiblissement équivalent pour la netteté; Reference System for determining Articulation Ratings*) filter (see column 4 of Table 3.1).

Because the P.48 IRS weighting used to be considered to model of an average narrow-band telephone handset used in the PSTN, the IRS weighting has been used to simulate speech signals obtained from a regular handset. Examples of standardization efforts using the P.48 weighting characteristic are the ITU-T Recommendations G.711, G.721, and G.728. This weighting, as defined in P.48, is sometimes called “full-IRS” weighting.

While the weighting characteristic in P.48 was considered to model connections over analog transmission facilities in the past (although it is not clear why the SRAEN filter should be included in both the send and receive paths), it is no longer representative of connections over modern digital facilities. In particular, the low frequency roll-off gives

Table 3.1: Send and receive amplitude frequency characteristics for the IRS response as in ITU-T Rec.P.48, the SRAEN filter, and the modified IRS (P.48 IRS with SRAEN filter insertion loss removed).

Frequency (Hz)	P.48 IRS		SRAEN Filter (dB)	Modified IRS	
	Send (dbPa/V)	Receive (dbPa/V)		Send (dbPa/V)	Receive (dbPa/V)
100	-45.8	-27.2	14.1	-31.7	-13.4
125	-36.1	-18.8	11.4	-24.7	-7.4
160	-25.6	-10.8	8.4	-17.2	-2.4
200	-19.2	-2.7	5.9	-13.3	3.2
250	-14.3	2.7	4.0	-10.3	6.7
300	-11.3	6.4	2.8	-8.5	9.2
315	-10.8	7.2	2.5	-8.3	9.7
400	-8.4	9.9	1.4	-7.0	11.3
500	-6.9	11.3	0.6	-6.3	11.9
600	-6.3	11.8	0.3	-6.0	12.1
630	-6.1	11.9	0.2	-5.9	12.1
800	-4.9	12.3	0.0	-4.9	12.3
1000	-3.7	12.6	0.0	-3.7	12.6
1250	-2.3	12.5	0.0	-2.3	12.5
1600	-0.6	13.0	0.1	-0.5	13.1
2000	0.3	13.1	-0.2	0.1	12.9
2500	1.8	13.1	-0.5	1.3	12.6
3000	1.5	12.5	0.5	2.0	13.0
3150	1.8	12.6	0.3	2.1	12.9
3500	-7.3	3.9	7.0	-0.3	10.9
4000	-37.2	-31.6	33.7	-3.5	2.1
5000	-52.2	-54.9	43.2	-9.0	-11.7
6300	-73.6	-67.5		-23*	
8000	-90.0	-90.0		-40*	
(*) : Values estimated from the modified IRS implemented in the STL.					

rise to unnecessary quality degradation. For the purpose of low bit-rate coder evaluation, especially where the coder is located in the telephone handset, a better characteristic can be obtained by modifying the P.48 full-IRS response to remove the SRAEN filter as shown in columns 5 and 6 of Table 3.1. These values are specified in Annex D of ITU-T Recommendation P.830 [7] and define the so-called “modified” IRS weighting. The modified IRS has been used in the development of ITU-T Recommendations G.723.1 and G.729, and will be used for the on-going standardization activities for an ITU-T 4 kbit/s speech codec.

The most important part of either the full or the modified IRS weighting is the transmission (or send) characteristic. The receive characteristic is less important because listening is in general done using handsets conforming to P.48 (which eliminates the need for filtering by the software, since it is done by the telephone terminal). In addition, the receive characteristic is relatively flat. Some studies also show that the use of headphones instead of handsets does not result in significantly different results while yielding lesser listener fatigue [8, 9]. Nevertheless, for cases where the receive-side MIRS filter is to be applied, a FIR implementation of this filter is available for 8000 Hz and 16000 Hz sampling rates.

An unspecified point in both P.48 and modified IRS is the phase response of the filter. There have been discussions within UGST on whether this should be implemented with a linear characteristic. The conclusion was that, since the phase response is unspecified, it should be kept as generic as possible, what is better accomplished by keeping the phase linear¹. If a certain non-linear phase characteristic is desired by the user, this can be implemented by cascading an all-pass filter with the desired phase response with one of the available FIR IRS implementations. Therefore, the IRS filters are implemented as FIR filters, as depicted in figure 3.1.

ITU-T Recommendation P.48 presents the nominal values for the amplitude response in column 2 of its Table 1 (here reproduced in column 2 of Table 3.1) and then the upper and lower tolerances listed in its Table 2. For the STL approach, it was decided to design IRS filters whose characteristic would deviate no more than 0.5 dB from the average values in P.48 (see in Figure 3.12 the agreement of the nominal values, repressed by dots, and the measured frequency response for the original P.48 IRS characteristic, represented by the continuous curve in the figure).

Other weightings

A filter that simulates the input response characteristic of certain mobile terminals was incorporated in the STL for data sampled at 16 kHz. Figures 3.10 and 3.11 display the respective frequency and impulse responses for the filter.

¹In spite of that, a non-linear phase IIR IRS filter is provided in the IIR module as an example of a cascade-form IIR filter implementation.

3.1.3 Wideband weighting

P.341 weighting

While the IRS filter is applicable to telephony bandwidth (or narrowband) speech, for wideband speech the specification for the send and receive sides is given in ITU-T Recommendation P.341 [10]. The mask specified in P.341 is rather wide, and an implementation of the send-side mask agreed on by the experts has been incorporated in the STL.

Other weightings

In the process to select a wideband codec at 32 and 24 kbit/s, a 5 kHz bandpass filter was developed and incorporated in the STL. Figures 3.22 and 3.23 display the respective frequency and impulse responses for the filter.

3.1.4 Noise weighting

Two weighting filters are available in this version of the STL, the psophometric and the Δ_{SM} weighting filters.

The psophometric weighting curve defined by ITU-T Recommendation O.41 is used for measuring the noise level in telephone circuits, accounting for the subjective perception of noise. The psophometric noise measure (given in dBmp) is related to the North-American C-message weighting curve (given in dBnC), using to the following:

$$dBmp = dBnC - 90.0dB$$

The other type of signal weighting filter is the Δ_{SM} , used for converting acoustic signals recorded in the far field using an omnidirectional microphone to the near-field equivalent of that signal if it were in the background of a telephone user. Owing to the directionality of the human mouth, head and torso, the high frequencies will mainly be radiated in the frontal direction, while the diffuse field will represent a spatial integration of the radiation in all directions [11]. Hence, the Δ_{SM} filter is deployed for weighting acoustic noises (babble, vehicular, etc.) before electrical summation with clean speech files, in order to simulate speech corrupted by background noise. It is useful in subjective listening tests where precise control of the actual SNR is necessary.

Both these filters have been implemented as FIR filters. The psophometric filter has been designed for speech sampled at 8 kHz, and the Δ_{SM} filter for speech sampled at 16 kHz. It should be noted that these filters, like the IRS filters, are also frequency-specific and, unlike the low-pass high-quality FIR filters described before, cannot be used for arbitrary rate ratio conversion.

3.1.5 PCM Quality

There are applications requiring the simulation of the response of filters found in the A/D and D/A interfaces of current transmission systems, which are in general PCM systems

satisfying ITU-T Recommendation G.711. The filters associated with G.711 are specified in Recommendation G.712 [12]. The main characteristic of these filters is the low out-of-band rejection of 25 dB.

In this context it is also necessary to simulate the conversion back to and forth the analog domain, e.g. to simulate multiple transcodings which are called *asynchronous transcodings*². One way to simulate asynchronous transcodings is by means of a non-linear phase filter (non-constant group delay), which is most efficiently implemented using IIR filters.

Infinite impulse response (IIR) filters used in this tool are of the parallel form (see figure 3.2), described by the equation:

$$H_I^p(z) = K + \sum_{l=1}^L \frac{b_{0l} + b_{1l}z^{-1}}{1 + c_{1l}z^{-1} + c_{2l}z^{-2}}$$

and of the cascade form (see figure 3.3), described by the equation:

$$H_I^c(z) = \prod_{l=1}^N \frac{b_{0l} + b_{1l}z^{-1} + b_{2l}z^{-2}}{1 + c_{1l}z^{-1} + c_{2l}z^{-2}}$$

3.2 Implementation

The rate change algorithm is organized in two modules, FIR and IIR, with prototypes respectively in `firflt.h` and `iirflt.h`. It evolved from a version initially developed by PKI, as part of the ETSI Half-rate GSM codec Host Laboratory exercise [13]. The rate-change functionality was incorporated in the STL92 in two main files, `hqflt.c` and `pcmflt.c`. To make these routines more flexible, the following modifications were included:

FIR: the FIR module was divided into a library source file (`fir-lib.c`) containing the basic filtering and initialization functions, as well as into source files for each kind of filter: `fir-flat.c` for high-quality low-pass and bandpass filters, `fir-irs.c` for the classical and modified IRS filters, and so on;

IIR: the IIR module was divided into a library file (`iir-lib.c`) containing basic filtering and initialization functions, as well as into source files for each kind of filter: `iir-g712.c` for G.712 filtering using the parallel-form filters, `iir-flat.c` for flat bandpass 1:3 and 3:1 asynchronization filtering using a cascade-form filter, and so on.

Files `fir-*.c` of the FIR module contain all the routines implementing FIR filters, i.e., the high-quality filters and IRS, Δ_{SM} and psophometric weighing filters. Files `iir-*.c`

²As the name indicates, there is no synchronisation between sampling instants of the two digital systems, i.e., re-sampling in the succeeding A/D is not synchronous to the clock in the preceding D/A converter.

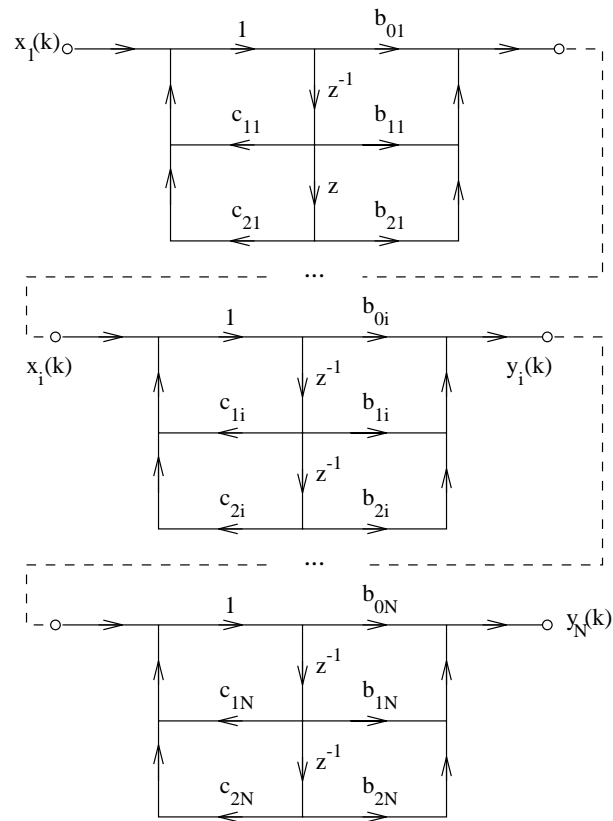


Figure 3.3: Cascade-form IIR filter block diagram.

of the IIR module implement the IIR filters, i.e., the parallel-form PCM filter and the cascade-form 3:1 asynchronization filter.

Some of these filters have been implemented using 24-bit coefficients, thus allowing real-time, bit-exact hardware implementation of these routines. It may be noted that, for these filters in the STL, the calculations are performed in floating point by converting the coefficients from the range $-2^{23}..2^{23}-1$ to $-1..+1$, which is not needed in real time hardware with fixed point DSPs.

Frequency response and impulse response plots are provided for the STL filters in the forthcoming sections. It should be noted, however, that the impulse responses shown have been computed from the 16-bit quantized impulse responses of the filters, as generated by the demonstration programs, while the frequency responses were calculated as described in section 3.3. It should be noted that the apparent asymmetry in some impulse responses happens because an integer number of samples are generated, and linear interpolation is used to draw the figure. If the impulse responses were derived directly from the filter coefficients, the plot would be symmetric.

NOTE: When the same filter type is used by several independent speech materials (e.g. several speech files) within the same execution of an application program, the user must remember that the filters have memory. Hence, wrong results can be obtained if a given number of initial samples are not discarded. See section 3.4 for an example, where the first 512 samples are skipped when calculating the power level of the output tone.

3.2.1 FIR module

The frequency responses of the implemented high-quality low-pass filters are shown in figures 3.4 and 3.5 (for rate-change factors 2 and 3, respectively), while the telephone bandwidth bandpass filter is given in figure 3.6 (only a rate-change factor of 2 is available). The impulse responses of these filters are given in figures 3.7, 3.8, and 3.9, respectively for the up-sampling filters (factors 2 and 3), for the down-sampling filters (factors 2 and 3), and for the bandpass filter.

The transmit-side IRS filter has been implemented for the “regular” and modified flavors. The regular transmit-side P.48 IRS filter amplitude responses are shown in figure 3.12 (the available sampling rates are 8 and 16 kHz). The transmit-side modified IRS filter is available for sampling at 16 kHz and 48 kHz, and their frequency responses are shown in figure 3.14. The impulse response of these transmit-side IRS filters are in figures 3.13 and 3.15 for the regular and modified IRS filters, respectively. The receive-side modified IRS filter has also been implemented and the frequency responses for 8 kHz and 16 kHz sampling rate are found in 3.16. The impulse responses of the receive-side modified IRS filters are shown in figure 3.17.

The frequency response of the STL psophometric filter is given in figure 3.18, and that of the Δ_{SM} filter in figure 3.19.

For wideband signals, two weighting filters are available. The transmit-side ITU-T P.341 filter amplitude response is shown in figure 3.20, and its impulse response is shown in figure

3.21. Alternatively to the P.341 filter, the frequency and impulse responses of the 5 kHz bandpass filter are shown in figures 3.22 and 3.23, respectively.

The high-quality filters were implemented for rate-change factors of 2 and 3. The IRS filters have been designed for specific *sampling rates* (e.g. 8 and 16 kHz). It should be noted that, while the high-quality filters are independent of the rate, the IRS filters are not, because the IRS mask is specified in terms of Hz, rather than normalized frequencies. This means that to carry out a high-quality up-sampling from 8 to 16 kHz, and from 16 to 32 kHz, the same routines are called, while for IRS there is no rate-change routine from 16 to 32 kHz.

Since the digital filters have memory, state variables are needed. In this version of the STL, a type **SCD_FIR** is defined, containing the past sample memory, as well as filter coefficients and other control variables. Its fields, whose values shall never be changed by the user, are as follows:

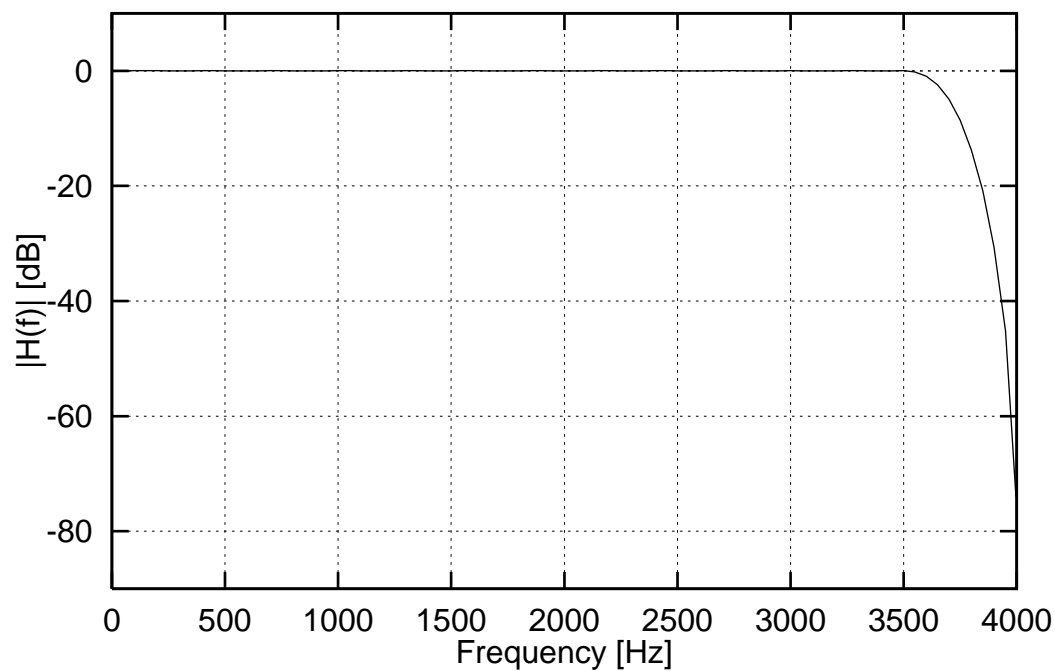
```

lenh0 .....Number of FIR coefficients
dwn_up ..... Down-sampling factor
k0 .....Start index in next segment (needed in segment-wise filtering)
h0 .....Pointer to array with FIR coefficients
T ..... Pointer to delay line
hswitch .....Switch to FIR-kernel: up- or down- sampling

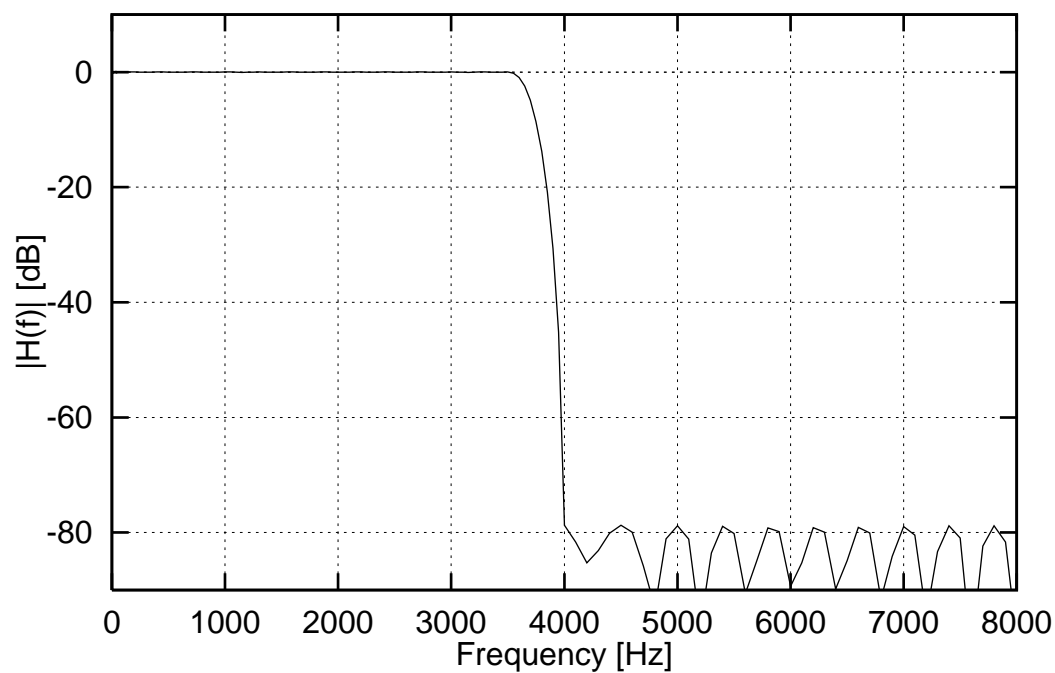
```

The relevant routines for each module are described in the next sections.³

³It should be noted that in the source code files there are local (privately-defined) functions which are not intended to be directly accessed by the user and therefore are not described here.

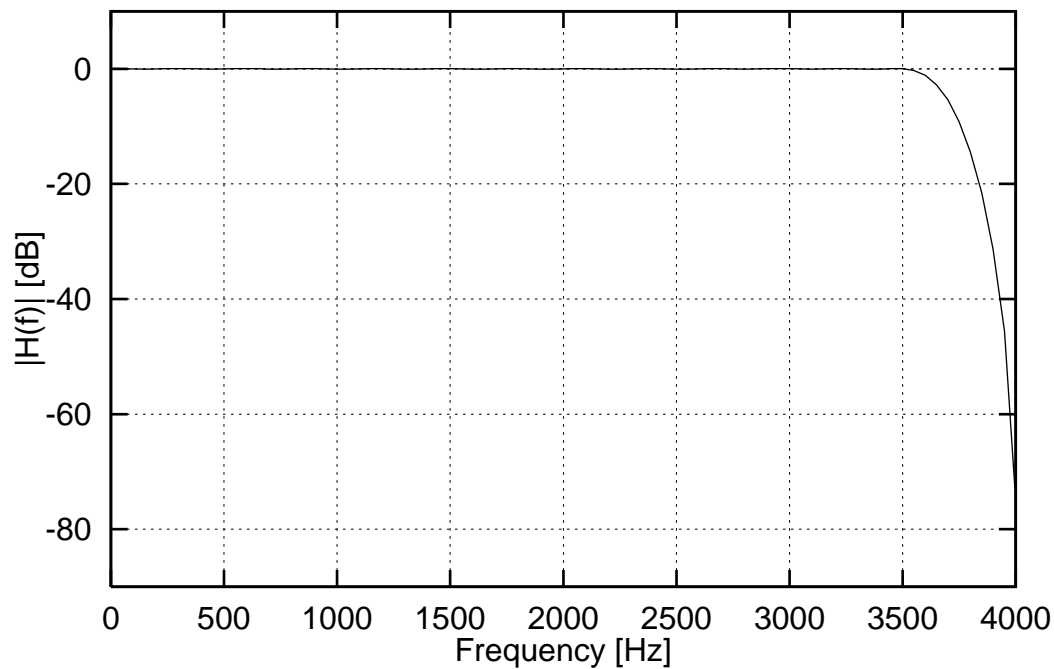


(a) High-quality filter for up-sampling.

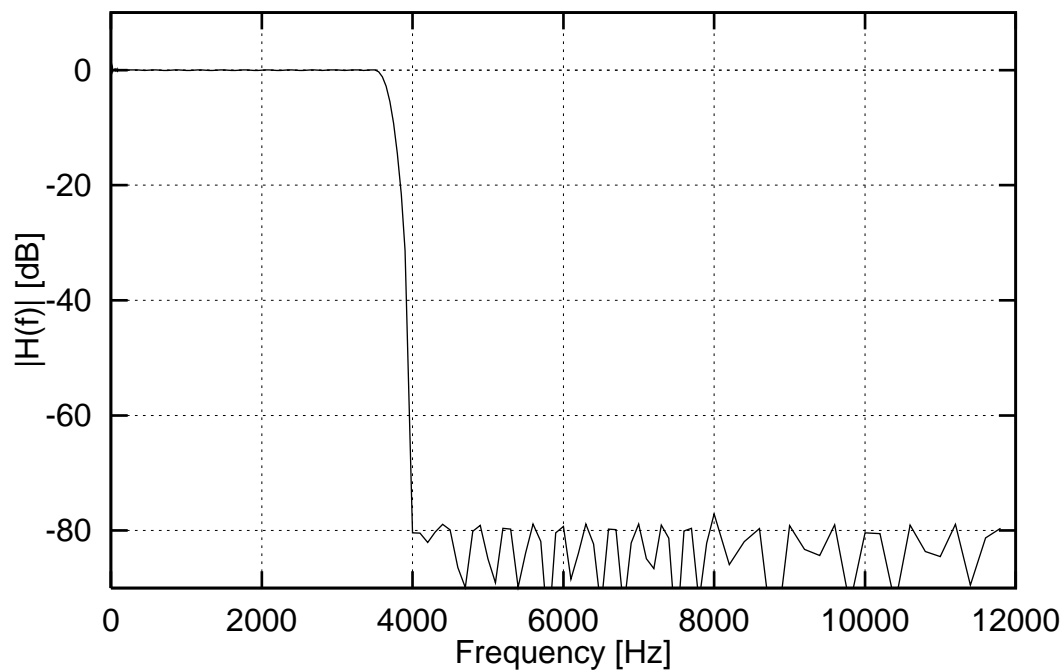


(b) High-quality filter for down-sampling.

Figure 3.4: High-quality filter responses for a factor of 2 and sampling rates of 8000 and 16000 Hz.

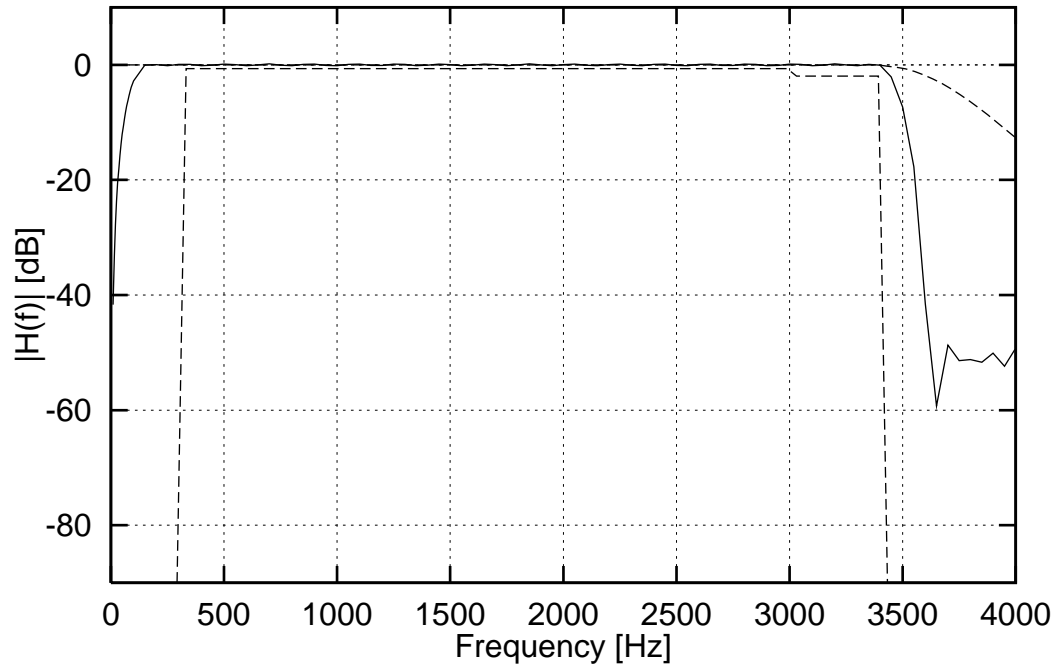


(a) High-quality filter for up-sampling.

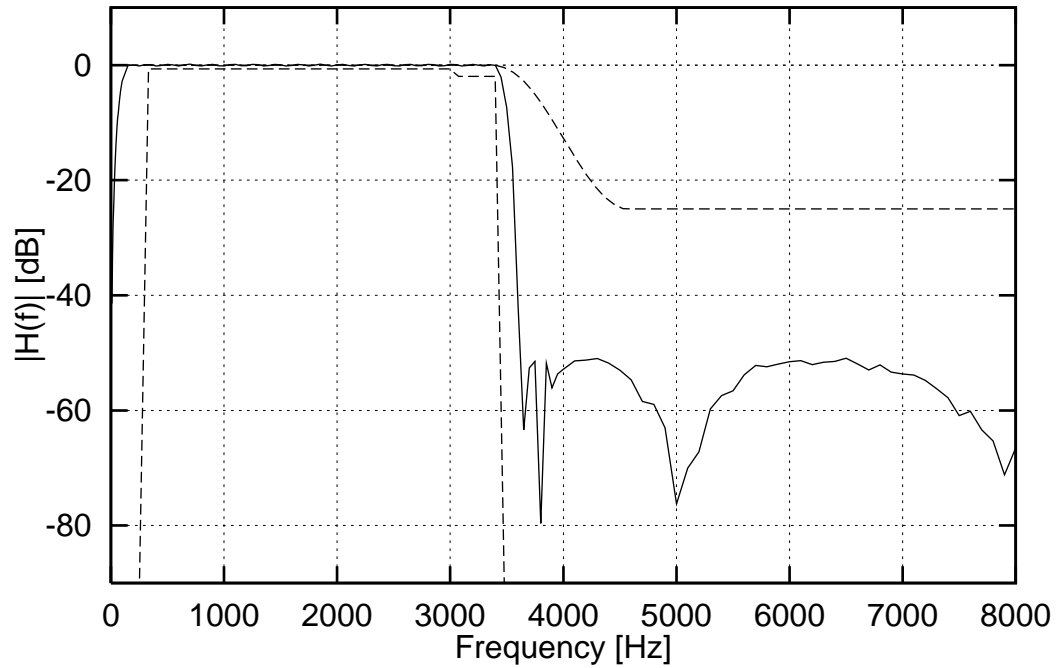


(b) High-quality filter for down-sampling.

Figure 3.5: High-quality filter responses for a factor of 3 and sampling rates of 8000 and 24000 Hz.



(a) High-quality bandpass for up-sampling (factor 1:2).



(b) High-quality bandpass for 2:1 down-sampling or for 1:1 filtering.

Figure 3.6: High-quality bandpass filter responses. Mask shown is that of the G.712 filter, for reference.

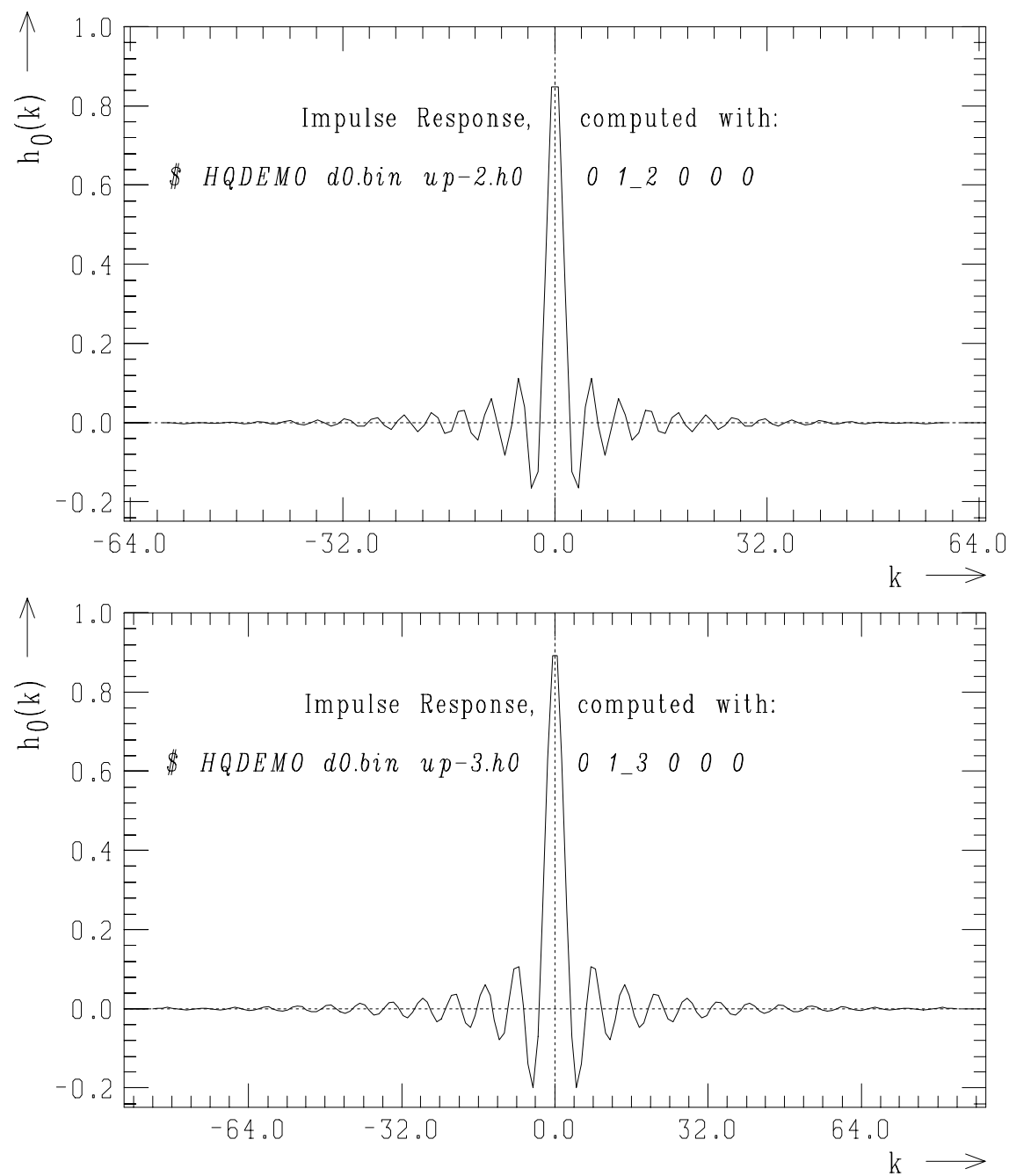


Figure 3.7: Impulse response for high-quality up-sampling filters (top, factor of 2; bottom, factor of 3).

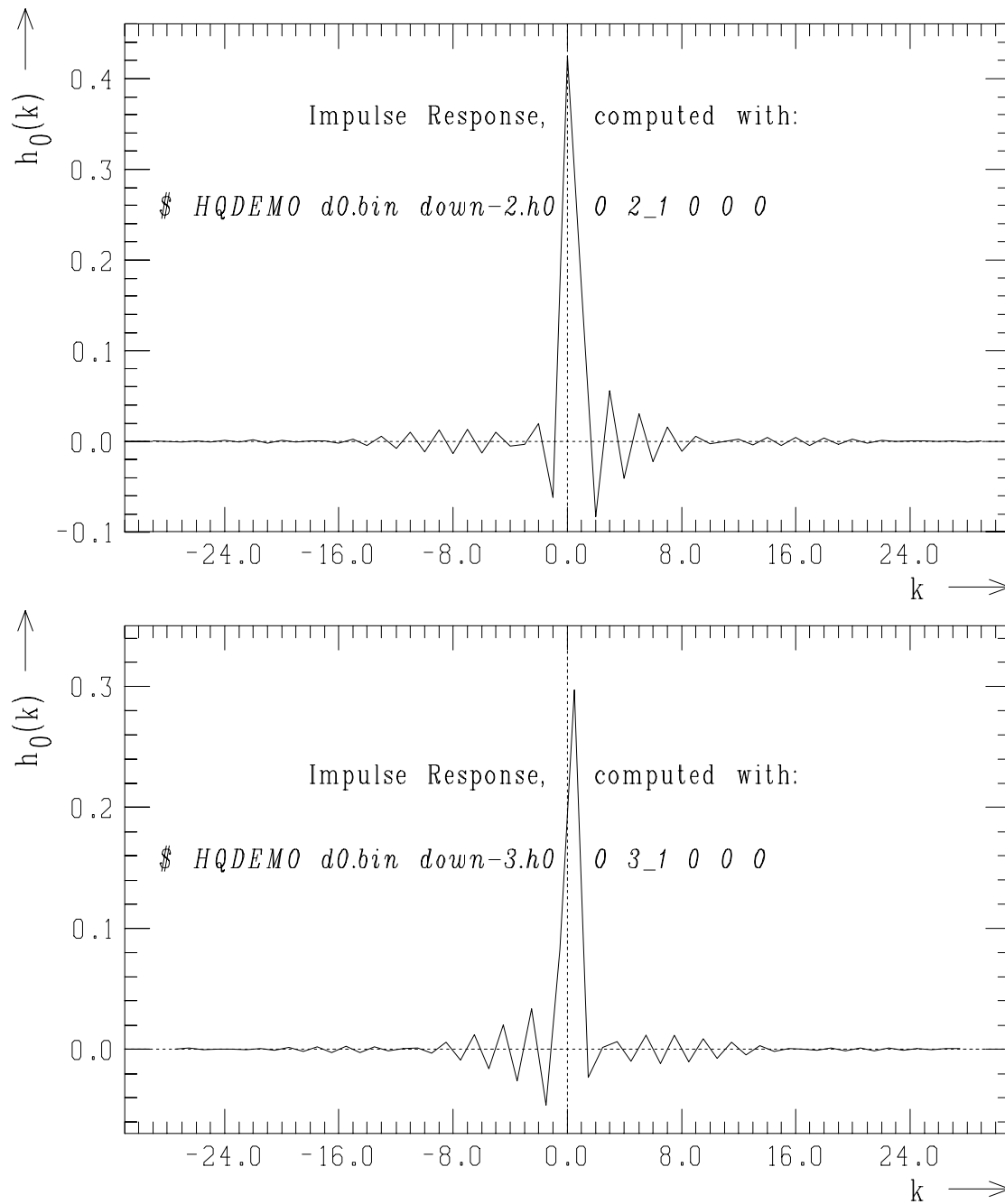
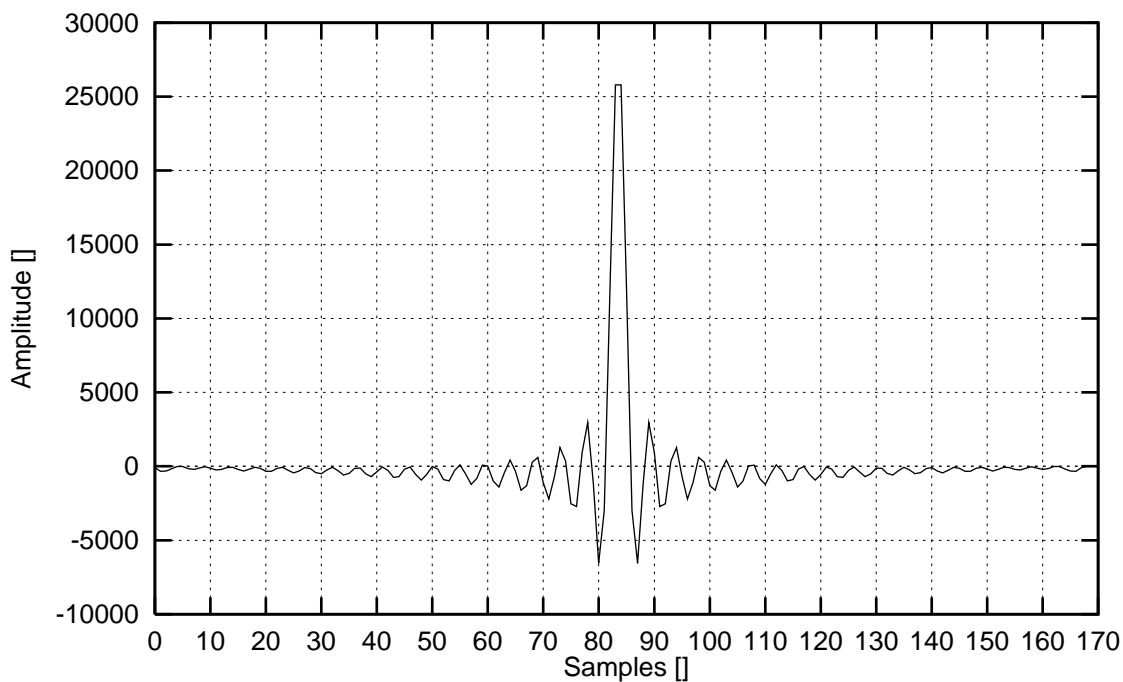
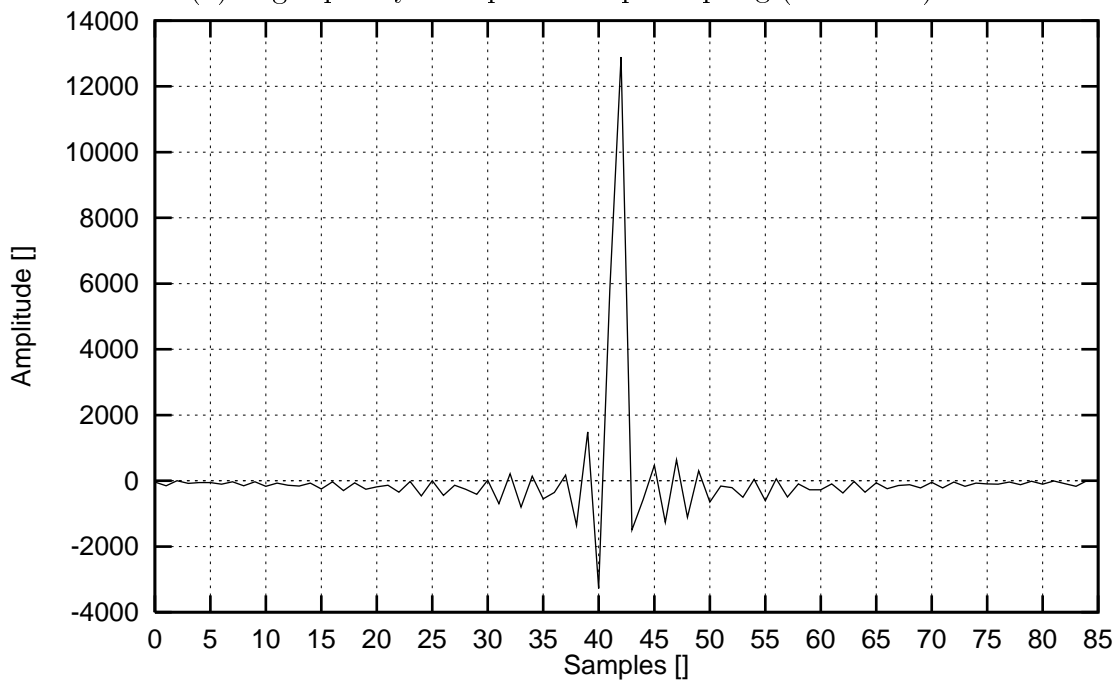


Figure 3.8: Impulse response for high-quality down-sampling filters (top, factor of 2; bottom, factor of 3).



(a) High-quality bandpass for up-sampling (factor 1:2).



(b) High-quality bandpass for down-sampling (factor 2:1).

Figure 3.9: Impulse response for high-quality bandpass filter (factors 2:1 and 1:1). Top is up-sampling by a factor of 1:2, and bottom is down-sampling by a factor of 2:1.

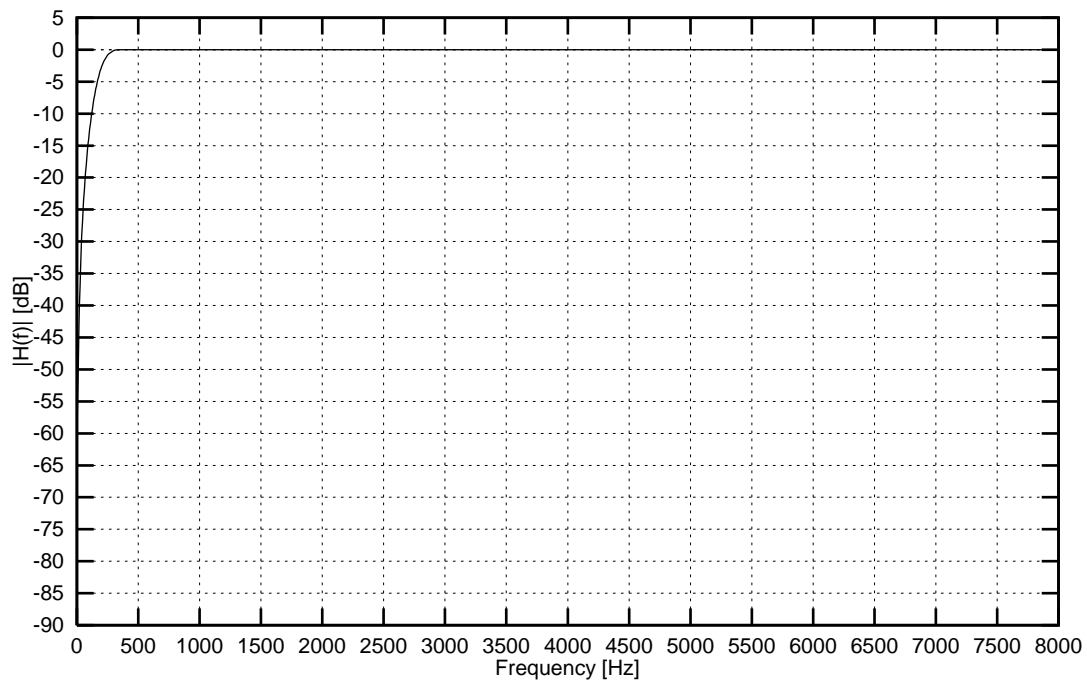


Figure 3.10: STL mobile station input (MSIN) frequency response for data sampled at 16 kHz (factor 1:1).

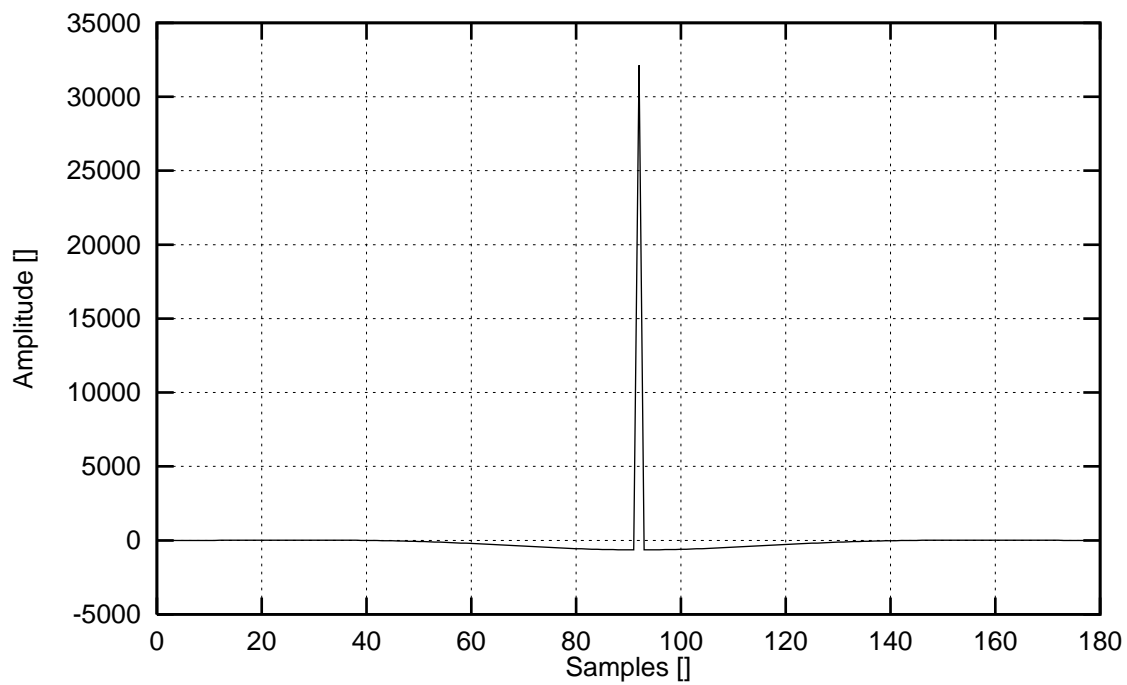


Figure 3.11: STL MSIN send-side filter impulse response.

3.2.1.1 *_init for the FIR module

Syntax:

```
#include "firflt.h"
SCD_FIR *delta_sm_16khz_init (void);
SCD_FIR *hq_down_2_to_1_init (void);
SCD_FIR *hq_up_1_to_2_init (void);
SCD_FIR *hq_down_3_to_1_init (void);
SCD_FIR *hq_up_1_to_3_init (void);
SCD_FIR *irs_8khz_init (void);
SCD_FIR *irs_16khz_init (void);
SCD_FIR *linear_phase_pb_2_to_1_init (void);
SCD_FIR *linear_phase_pb_1_to_2_init (void);
SCD_FIR *linear_phase_pb_1_to_1_init (void);
SCD_FIR *msin_16khz_init();
SCD_FIR *mod_irs_16khz_init (void);
SCD_FIR *mod_irs_48khz_init (void);
SCD_FIR *rx_mod_irs_8khz_init(void);
SCD_FIR *rx_mod_irs_16khz_init(void);
SCD_FIR *psophometric_8khz_init (void);
SCD_FIR *p341_16k_init (void);
SCD_FIR *bp5k_16k_init (void);
```

Prototypes: firflt.h

Description:

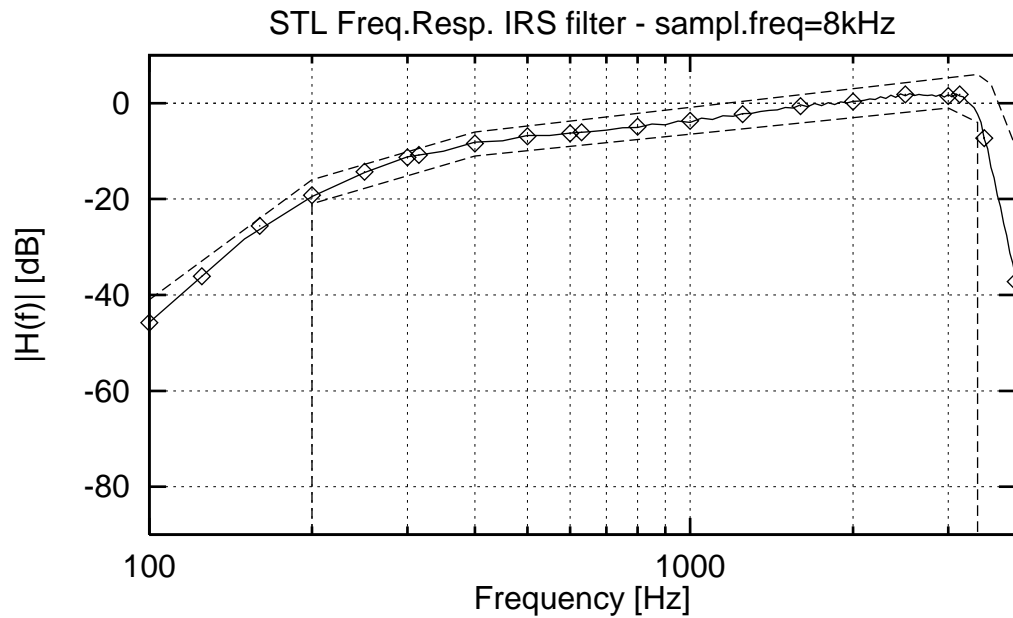
`delta_sm_16khz_init` is the initialization routine for the Δ_{SM} weighting filter for data sampled at 16 kHz using a linear phase FIR filter structure. Input and output signals will be at 16 kHz. Code is in file `fir-dsm.c` and its frequency response is given in figure 3.19.

`hq_up_1_to_2_init` is the initialization routine for high quality FIR up-sampling filtering by a factor of 2. The -3 dB point for this filter is located at approximately 3660 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in figures 3.4(a) and 3.7 (top), respectively.

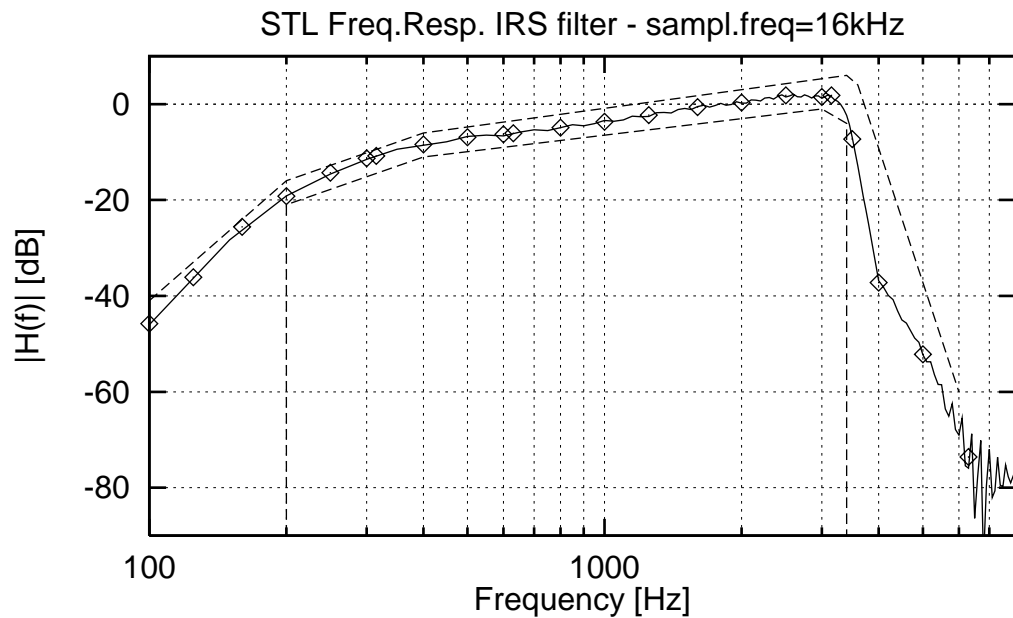
`hq_down_2_to_1_init` is the initialization routine for high quality FIR down-sampling filtering by a factor of 2. The -3 dB point for this filter is located at approximately 3660 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in figures 3.4(b) and 3.8 (top), respectively.

`hq_up_1_to_3_init` is the initialization routine for high quality FIR up-sampling filter by factor of 3. The -3 dB point for this filter is located at approximately 3650 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in figures 3.5(a) and 3.7 (bottom), respectively.

`hq_down_3_to_1_init` is the initialization routine for high quality FIR down-sampling filtering by a factor of 3. The -3 dB point for this filter is located at approximately 3650



(a) Transmission-side IRS for input samples at 8 kHz.



(b) Transmission-side IRS for input samples at 16 kHz.

Figure 3.12: Transmission-side IRS filter responses. The diamonds represent the nominal values of the “full” IRS characteristic and the interrupted line represent the mask of the “full” IRS, as shown in figure 2 of ITU-T Rec. P.48.

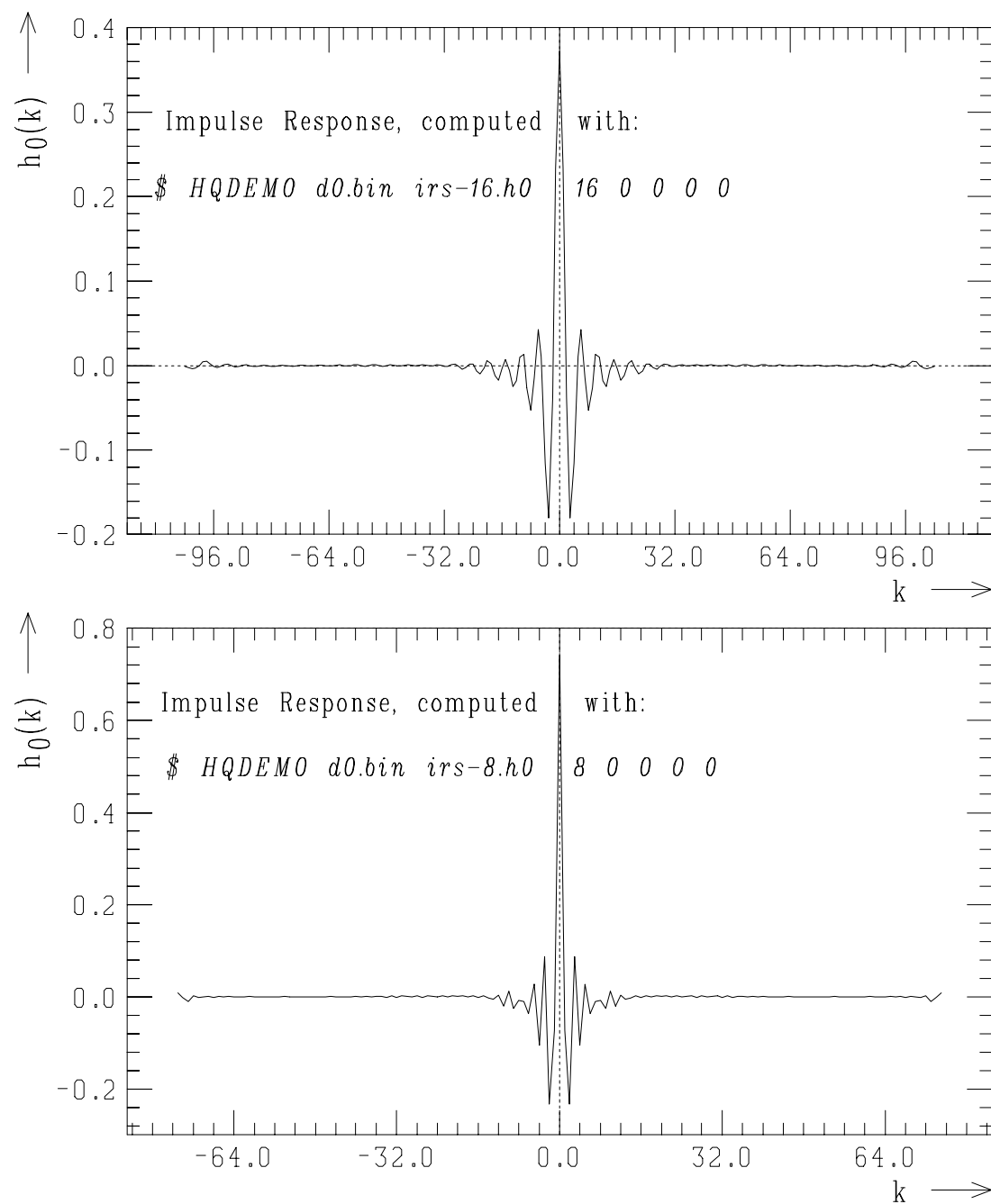
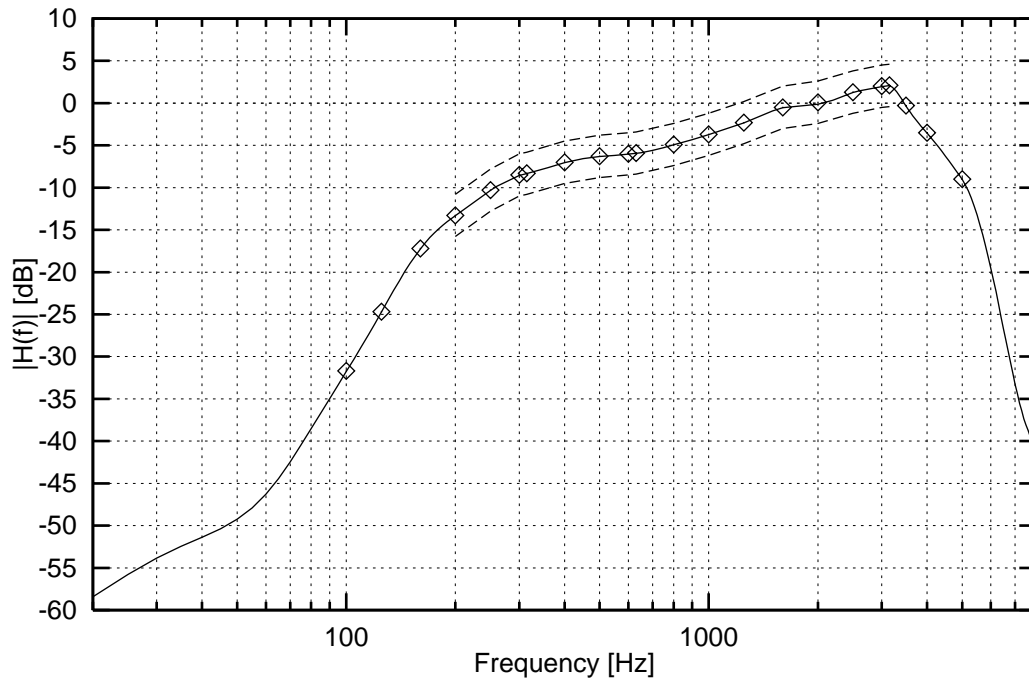
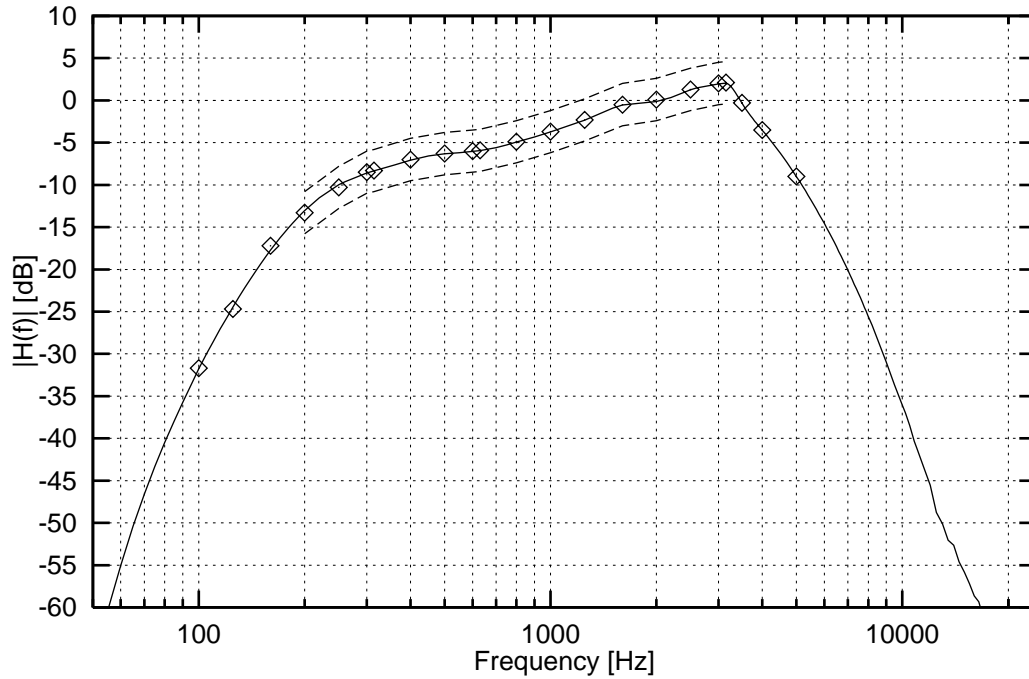


Figure 3.13: Impulse response of transmission-side IRS filters at 16 and 8 kHz.

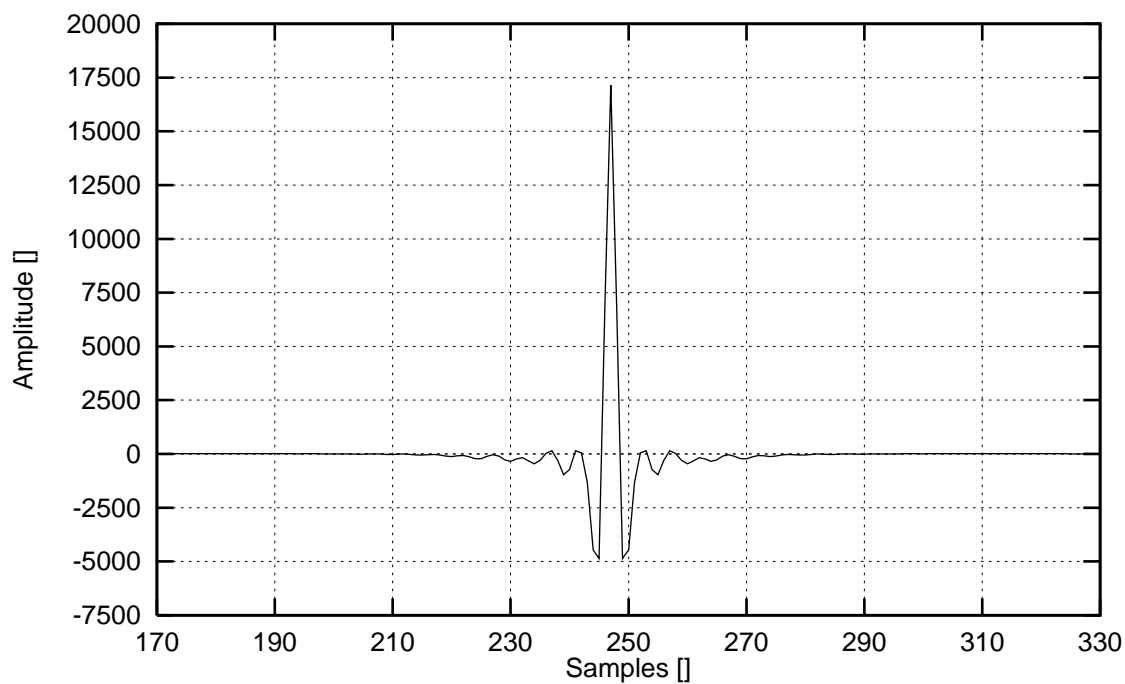


(a) Transmission-side modified IRS for input samples at 16 kHz.

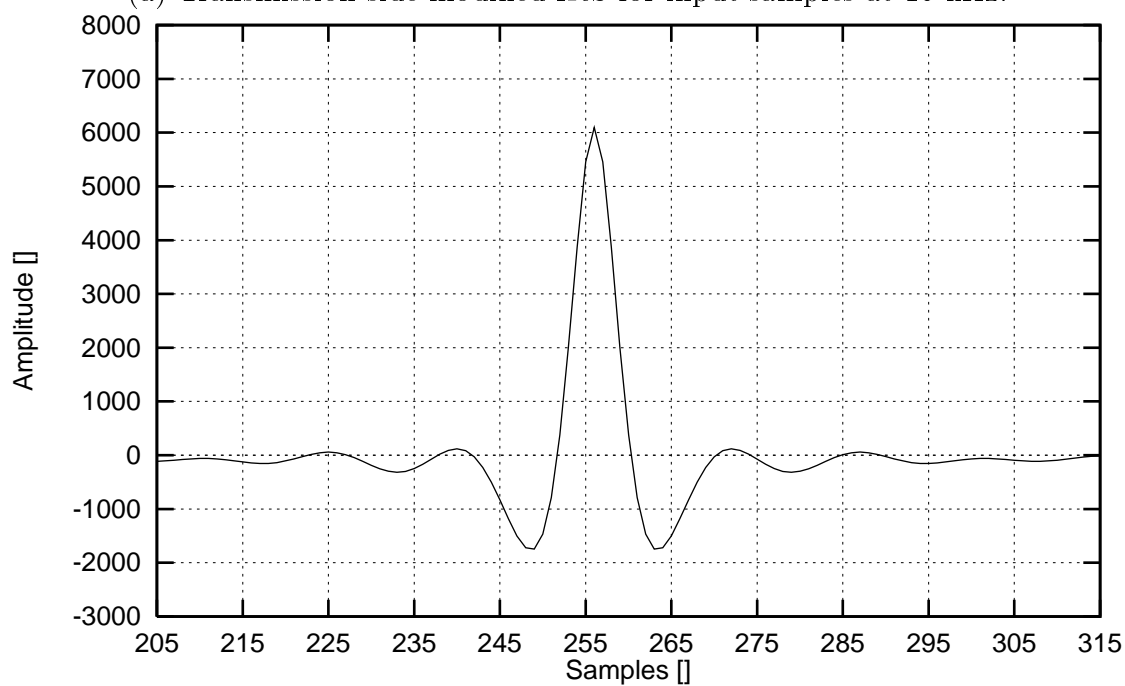


(b) Transmission-side modified IRS for input samples at 48 kHz.

Figure 3.14: Transmission-side modified IRS filter responses. The interrupted line represents the mask of the "full" IRS.

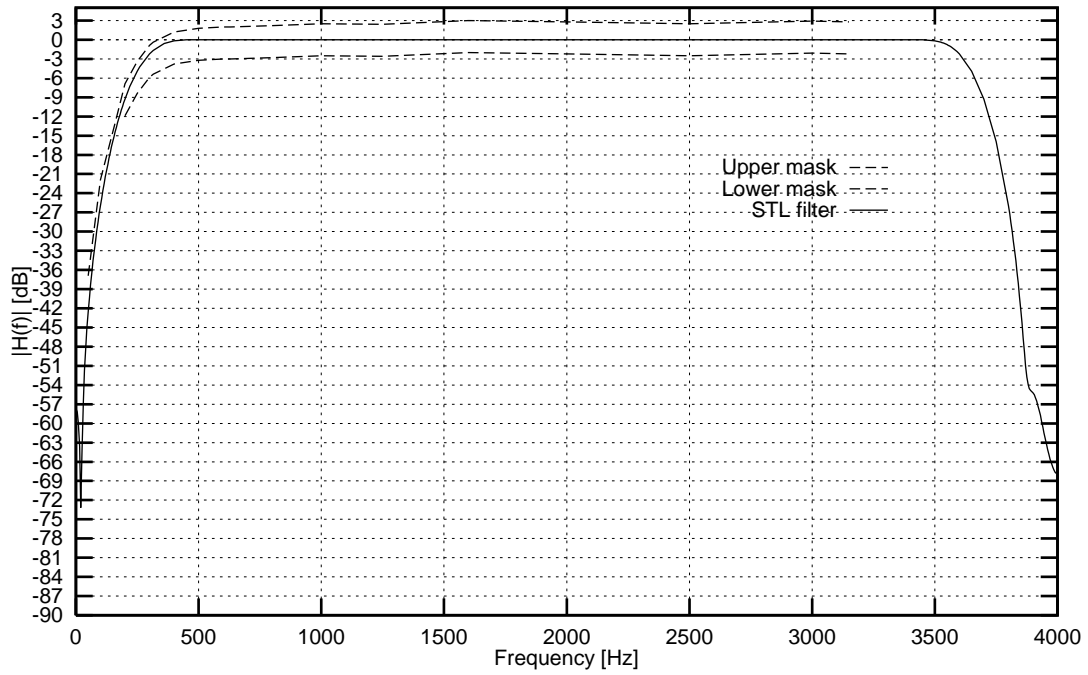


(a) Transmission-side modified IRS for input samples at 16 kHz.

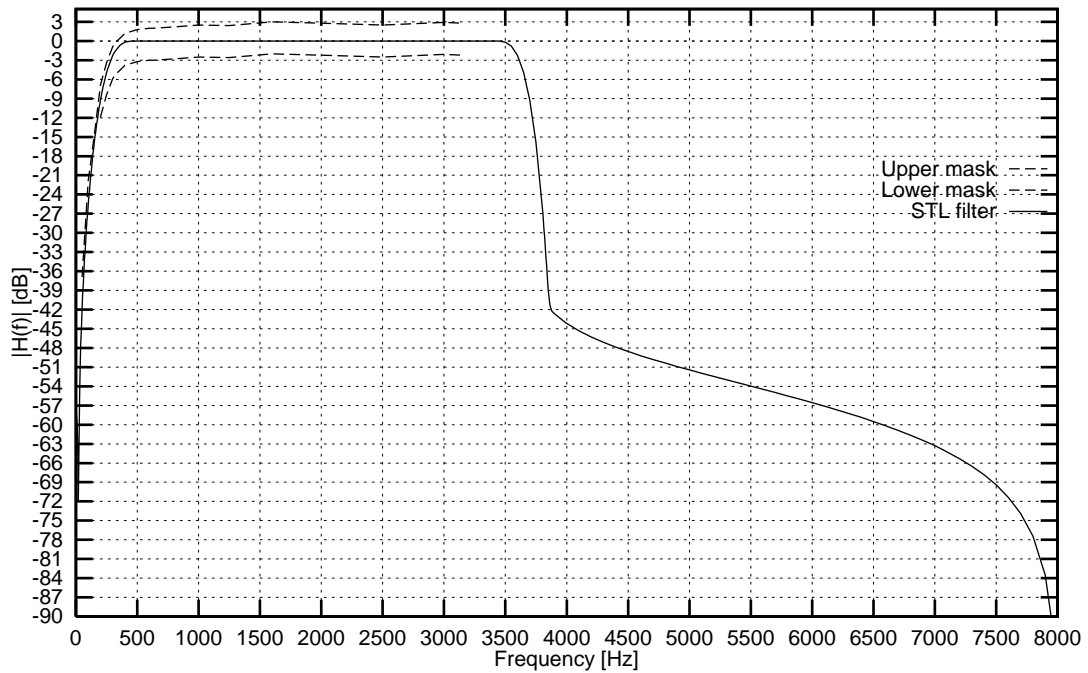


(b) Transmission-side modified IRS for input samples at 48 kHz.

Figure 3.15: Impulse response of transmission-side modified IRS filters at 16 kHz (top) and 48 kHz (bottom).

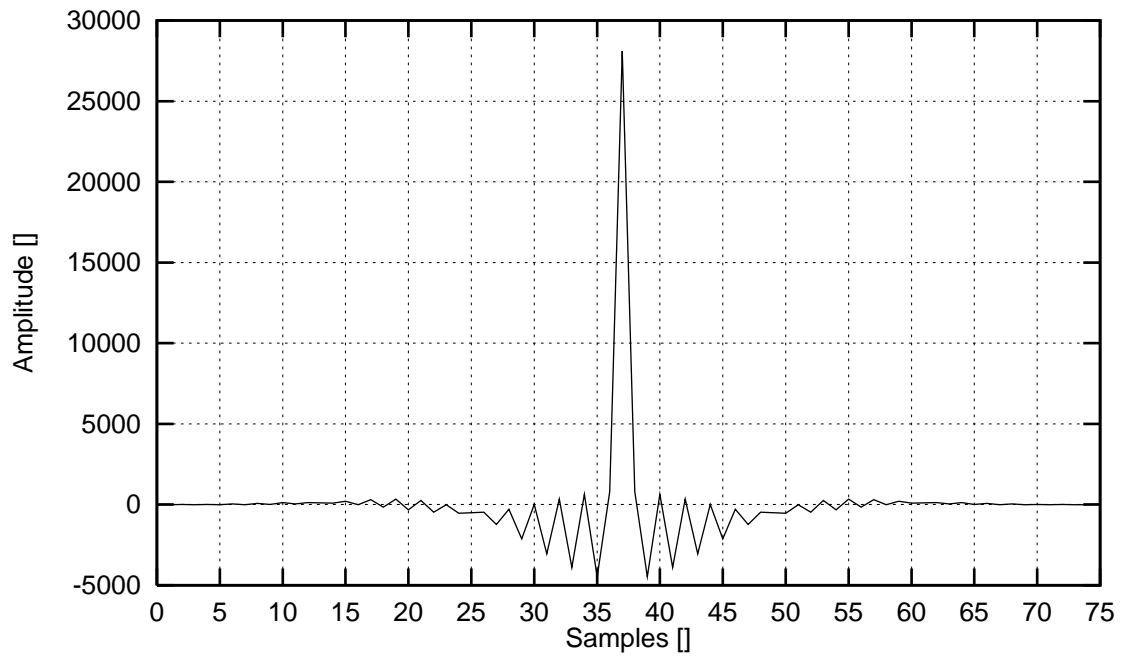


(a) Receive-side modified IRS for input samples at 8 kHz.

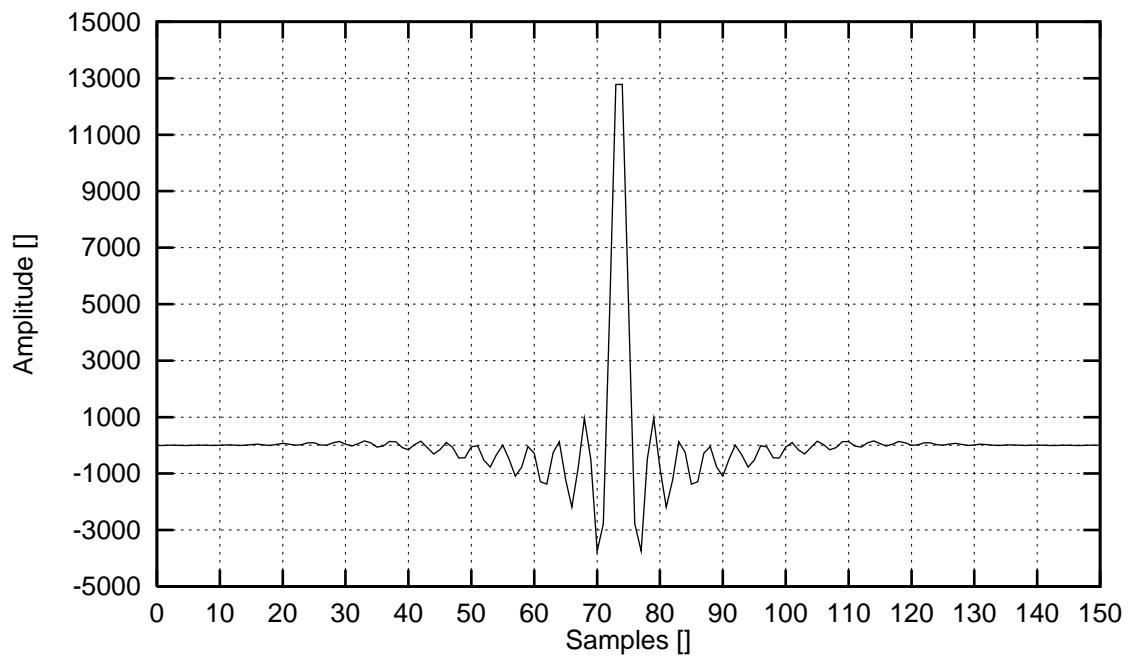


(b) Receive-side modified IRS for input samples at 16 kHz.

Figure 3.16: Receive-side modified IRS filter responses. The diamonds represent the nominal values of the modified IRS characteristic and the interrupted line represent the mask of the “full” IRS.



(a) Receive-side modified IRS for input samples at 8 kHz.



(b) Receive-side modified IRS for input samples at 16 kHz.

Figure 3.17: Impulse response of receive-side modified IRS filters at 8 kHz (top) and 16 kHz (bottom).

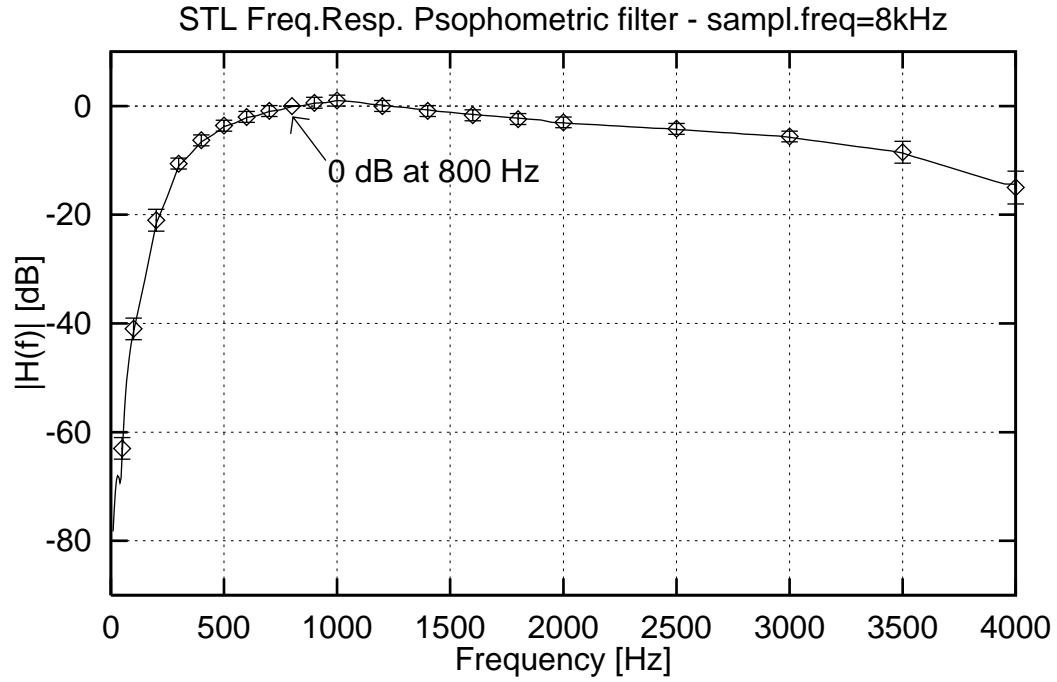


Figure 3.18: Frequency response for the psophometric filter. The points show the average points and the allowed range as per ITU-T Rec. O.41.

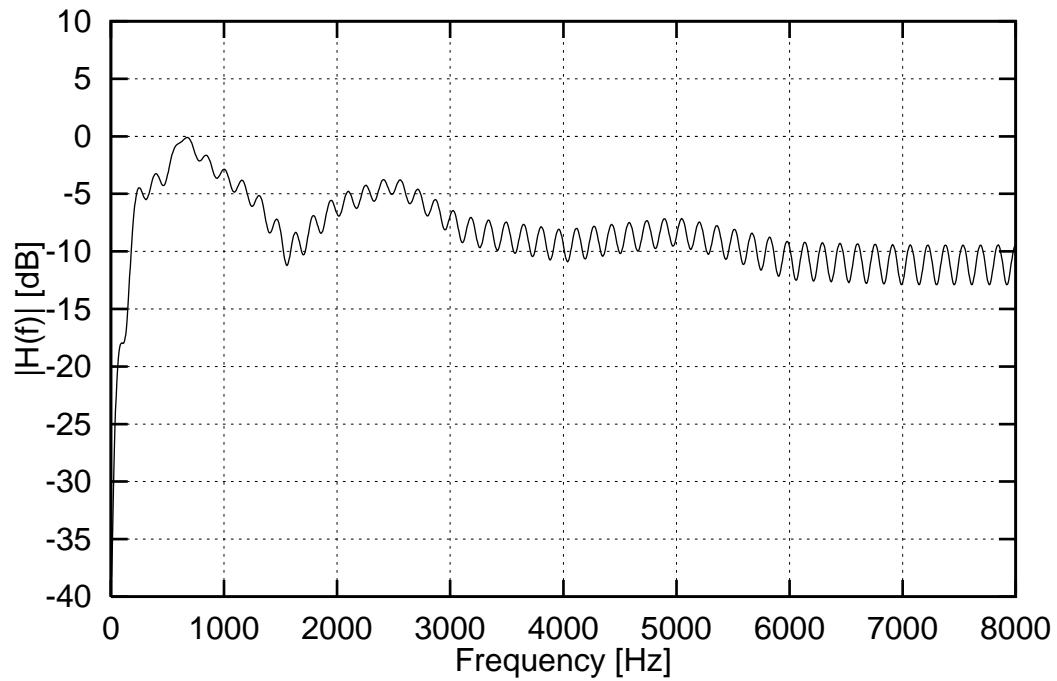


Figure 3.19: Frequency response for the Δ_{SM} filter.

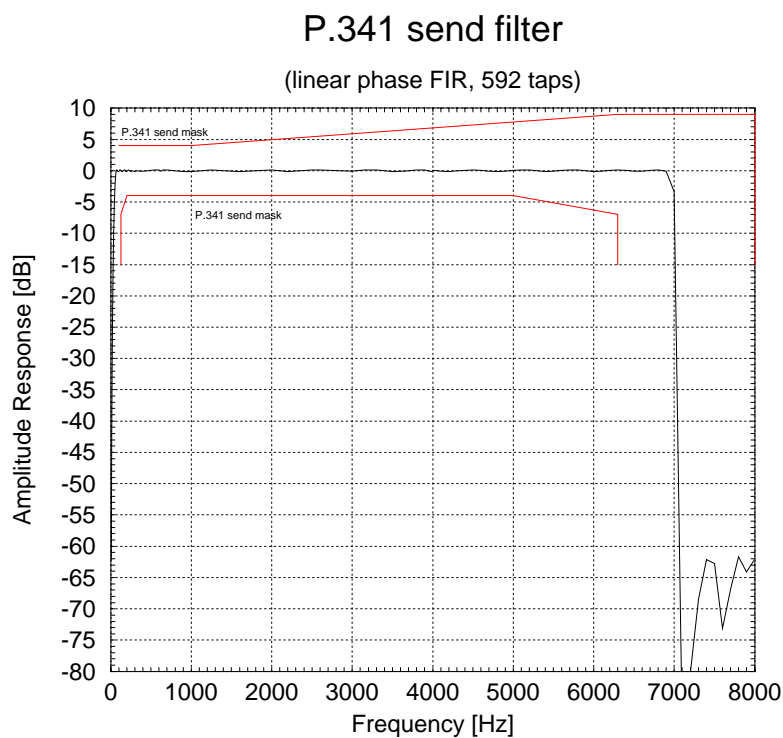


Figure 3.20: STL P.341 send-side filter frequency response for data sampled at 16 kHz (factor 1:1).

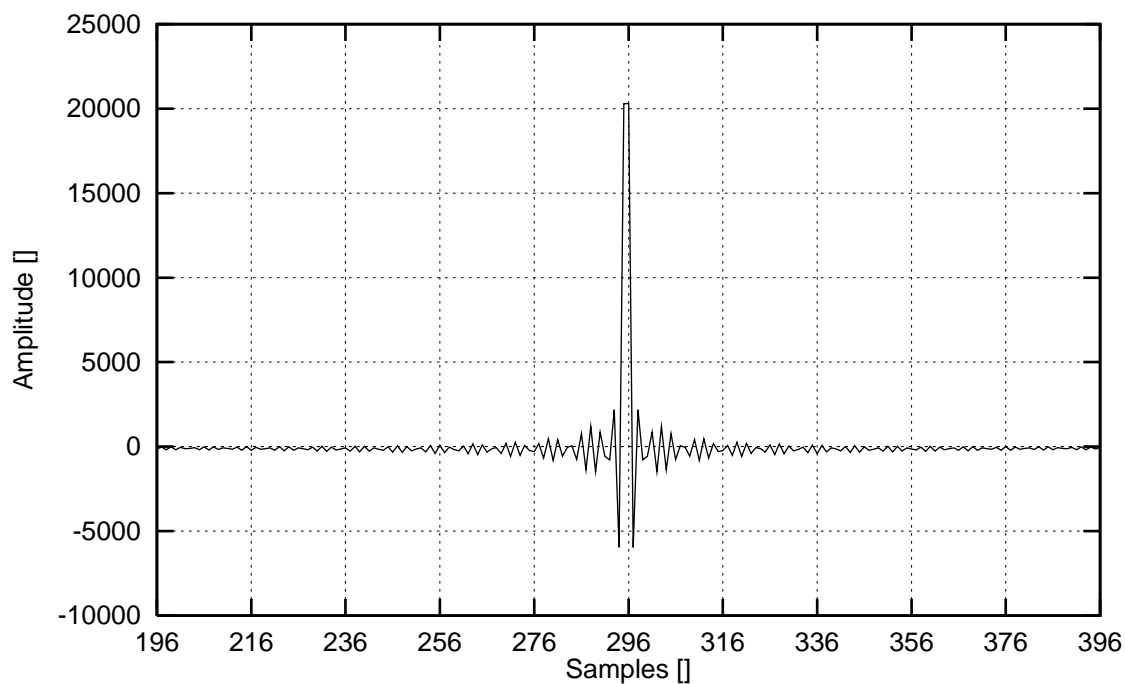


Figure 3.21: STL P.341 send-side filter impulse response.

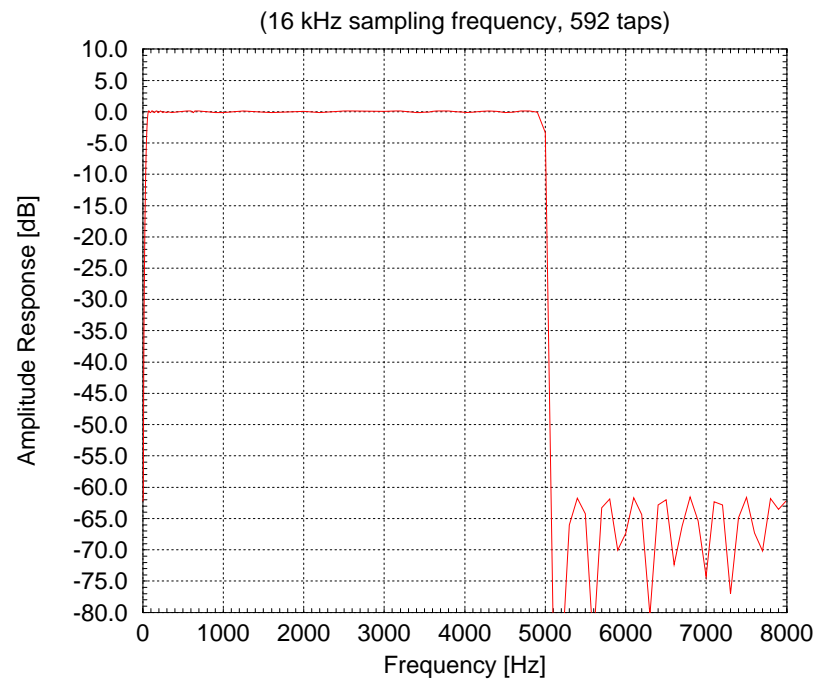


Figure 3.22: STL 5 kHz band limiting filter frequency response for data sampled at 16 kHz (factor 1:1).

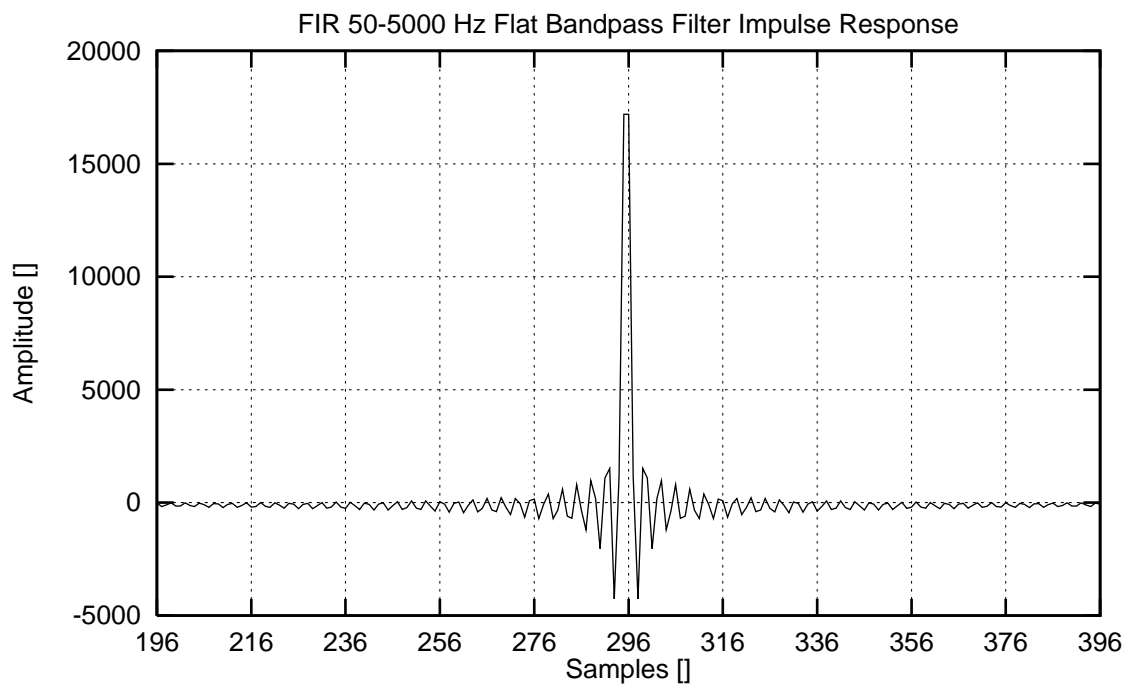


Figure 3.23: STL 5 kHz band limiting filter impulse response.

Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in figures 3.5(b) and 3.8 (bottom), respectively.

`linear_phase_bp_1_to_2_init` is the initialization routine for bandpass, FIR up-sampling filtering by a factor of 2. The -3 dB points for this filter are located at approximately 98 and 3460 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in figures 3.6(b) and 3.9(b), respectively.

`linear_phase_bp_2_to_1_init` is the initialization routine for bandpass, FIR down-sampling filtering by a factor of 2. The -3 dB points for this filter are located at approximately 98 and 3460 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in figures 3.6(a) and 3.9(a), respectively.

`linear_phase_bp_1_to_1_init` is the initialization routine for FIR 1:1 bandpass filtering. The -3 dB points for this filter are located at approximately 98 and 3460 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in figures 3.6(a) and 3.9(a), respectively.

`msin_16khz_init` is the initialization routine for the high-pass, FIR 1:1 filter that simulates a mobile station input characteristic. The -3 dB point for this filter is located at approximately 195 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in figures 3.10 and 3.11, respectively.

`irs_8khz_init` is the initialization routine for the transmit-side IRS weighting filter for data sampled at 8 kHz using a linear phase FIR filter structure. Input and output signals will be at 8 kHz. Code is in file `fir-irs.c` and its frequency and impulse response are given in figures 3.12 and 3.13, respectively.

`irs_16khz_init` is the initialization routine for the transmit-side IRS weighting filter for data sampled at 16 kHz using a linear phase FIR filter structure. Input and output signals will be at 16 kHz. Code is in file `fir-irs.c` and its frequency and impulse response are given in figures 3.12 and 3.13, respectively.

`mod_irs_16khz_init` is the initialization routine for the transmit-side modified IRS weighting filter for data sampled at 16 kHz using a linear phase FIR filter structure. Input and output signals will be at 16 kHz since no rate change is performed by this function. Code is in file `fir-irs.c` and its frequency and impulse response are given in figures 3.14 and 3.15, respectively.

`mod_irs_48khz_init` is the initialization routine for the transmit-side modified IRS weighting filter for data sampled at 48 kHz using a linear phase FIR filter structure. Input and output signals will be at 48 kHz since no rate change is performed by this function. Code is in file `fir-irs.c` and its frequency and impulse response are given in figures 3.14 and 3.15, respectively.

`rx_mod_irs_8khz_init` is the initialization routine for the receive-side modified IRS weighting filter for data sampled at 8 kHz using a linear phase FIR filter structure. The -3 dB points for this filter are located at approximately 285 Hz and 3610 Hz. Input and output signals will be at 8 kHz since no rate change is performed by this function. Code is in file `fir-irs.c` and its frequency and impulse response are given in figures 3.16 and 3.17, respectively.

`rx_mod_irs_16khz_init` is the initialization routine for the receive-side modified IRS

weighting filter for data sampled at 16 kHz using a linear phase FIR filter structure. The -3 dB points for this filter are located at approximately 285 Hz and 3610 Hz. Input and output signals will be at 16 kHz since no rate change is performed by this function. Code is in file `fir-irs.c` and its frequency and impulse response are given in figures 3.16 and 3.17, respectively.

`psophometric_8khz_init` is the initialization routine for the O.41 psophometric weighting filter for data sampled at 8 kHz using a linear phase FIR filter structure. Input and output signals will be at 8 kHz since no rate change is performed by this function. Code is in file `fir-pso.c` and its frequency response is given in figure 3.18.

`p341_16khz_init` is the initialization routine for the P.341 send-side weighting filter for data sampled at 16 kHz. Input and output signals will be at 16 kHz since no rate change is performed by this function. Its frequency response is shown in figure 3.20 and its impulse response is shown in figure 3.21. The -3 dB points for this filter are located at approximately 50 and 7000 Hz. Code is in file `fir-wb.c`.

`bp5k_16khz_init` is the initialization routine for a 5-kHz-band limiting of wideband signals sampled at 16 kHz. Input and output signals will be at 16 kHz since no rate change is performed by this function. Its frequency response is shown in figure 3.22 and its impulse response is shown in figure 3.23. The -3 dB points for this filter are located at approximately 50 and 4990 Hz. Code is in file `fir-wb.c`.

Variables:

None.

Return value:

These functions return a pointer to a state variable structure of type `SCD_FIR`.

3.2.1.2 `hq_kernel`

Syntax:

```
#include "firflt.h"
long hq_kernel(long lseg, float *x_ptr, SCD_FIR *fir_ptr, float *y_ptr);
```

Prototype: `firflt.h`

Source code: `fir-lib.c`

Description:

This is the main entry routine for generic FIR filtering. It works as a switch to specific up- and down-sampling FIR-kernel functions. The adequate lower-level filtering routine private to the filtering module (which is not visible by the user) is defined by the initialization routines. Currently, this function does not work properly for sample-by-sample downsampling operation, i.e. when `lseg = 1`. This limitation should be corrected in a future version.

Please note that prior to the first call to `hq_kernel`, one of the initialization routines `hq*_init` must be called to allocate memory for state variables and to set the desired

filter coefficients.

After returning from this function, the state variables are saved to allow segment-wise filtering through successive calls of `hq_kernel`. This is useful when large files have to be processed.

Variables:

<i>lseg</i>	Number of input samples. Should be larger than 1 for proper downsampling operation.
<i>x_ptr</i>	Array with input samples.
<i>fir_ptr</i>	Pointer to FIR-struct.
<i>y_ptr</i>	Pointer to output samples.

Return value:

The number of filtered samples as a `long`.

3.2.1.3 `hq_reset`

Syntax:

```
#include "firflt.h"
void hq_reset (SCD_FIR *fir_ptr);
```

Prototype: `firflt.h`

Source code: `fir-lib.c`

Description:

Clear state variables in `SCD_FIR` struct; deallocation of filter structure memory is not done. Please note that *fir_ptr* should point to a valid `SCD_FIR` structure, which was allocated by an earlier call to one of the FIR initialization routines `hq*_init`.

Variables:

<i>fir_ptr</i>	Pointer to a valid structure <code>SCD_FIR</code> .
----------------	-------	---

Return value:

None.

3.2.1.4 `hq_free`

Syntax:

```
#include "firflt.h"
void hq_free (SCD_FIR *fir_ptr);
```

Prototype: `firflt.h`

Source code: `fir-lib.c`

Description:

Deallocate memory, which was allocated by an earlier call to one of the FIR initialization routines `hq*_init`. Note that the pointer to the structure `SCD_FIR` must not be a null

pointer.

Variables:

fir_ptr Pointer to a structure of type `SCD_FIR`.

Return value:

None.

3.2.2 IIR Module

The IIR module contains filters whose main use is for asynchronous filtering. For telephony bandwidth asynchronous filtering, PCM filters are available in both cascade and parallel IIR filter forms. For wideband speech (50–7000 Hz), 3:1 and 1:3 rate-change factor filters are available. A transmit-side IRS filter for speech sampled at 8 kHz is also available in this module as an example of implementation of an IIR cascade-form filter.

The PCM filters have been designed for *sampling rates* of 8 and 16 kHz. It should be noted that the G.712 mask is specified in terms of Hz, rather than normalized frequencies. Therefore this applies only to rate conversions of factor 2, i.e., 8 kHz to 16 kHz and 16 kHz to 8 kHz. The frequency responses of the implemented PCM filters are shown in figure 3.28.

Since the digital filters need memory, state variables are needed. In the STL, a type `SCD_IIR` has been defined for parallel-form IIR filters, containing the past memory samples as well as filter coefficients and other control variables. Its fields are as follows:

nblocks Number of coefficient sets
idown Up-/down-sampling factor
k0 Start index in next segment
gain Gain factor
direct_cof Direct path coefficient
b[3] Pointer to numerator coefficients
c[2] Pointer to denominator coefficients
T[2] Pointer to state variables
hswitch Switch to IIR-kernel: Up or down-sampling

For the cascade-form IIR filters, the state variable structure defined is `CASCADE_IIR` which is slightly different from the one for the parallel form structure:

nblocks Number of stages in cascade
idown Up-/down-sampling factor
k0 Start index in next segment
gain Gain Factor
a[2] Pointer to numerator coefficients
b[2] Pointer to denominator coefficients
T[4] Pointer to state variables
hswitch Switch to IIR-kernel: Up or down-sampling

It should be noted that the values of the fields must not be altered, and for most purposes they are not needed by the user. The relevant routines for each module are described in

the next sections.

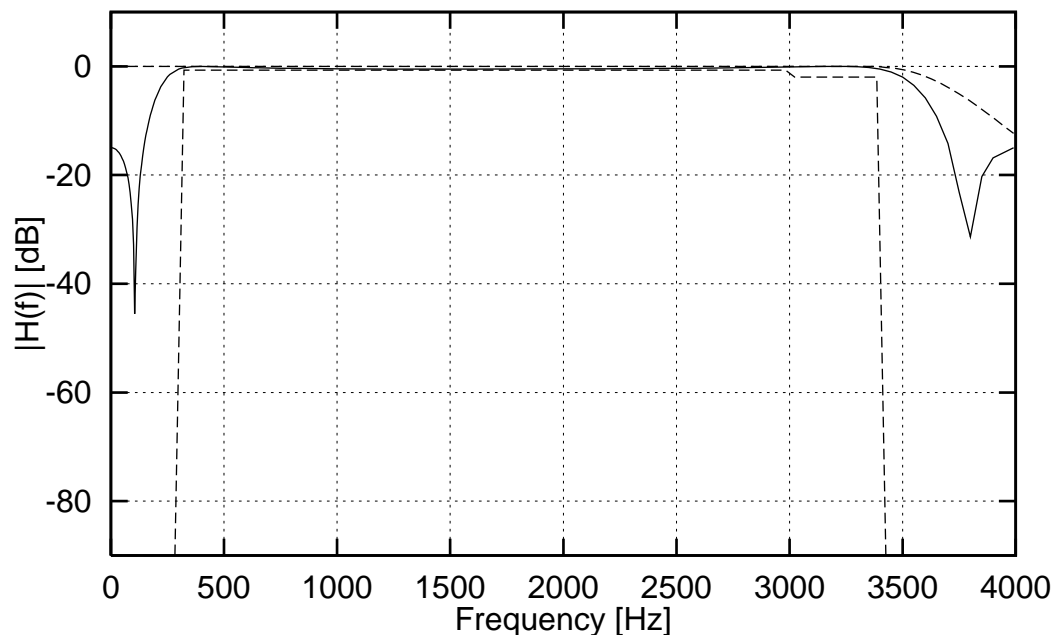


Figure 3.24: Frequency response of the cascade implementation of the G.712 standard PCM filter for data sampled at 8 kHz.

3.2.2.1 iir*_init

Syntax:

```
#include "iirflt.h"
CASCADE_IIR *iir_G712_8khz_init (void);
CASCADE_IIR *iir_irs_8khz_init (void);
CASCADE_IIR *iir_casc_lp_3_to_1_init(void);
CASCADE_IIR *iir_casc_lp_1_to_3_init(void);
```

Prototypes: iirflt.h

Description:

`iir_G712_8khz_init` initializes an 8 kHz cascade IIR filter structure for a standard PCM (G.712) filtering. Input and output signals will be at 8 kHz since no rate change is performed by this function. The -3 dB points for this filter are located at approximately 230 and 3530 Hz. Its source code is found in file `cascg712.c` and its frequency response is given in figure 3.24.

`iir_irs_8khz_init` initializes an 8 kHz cascade IIR filter structure for a transmit-side P.48 IRS non-linear phase filtering. Input and output signals will be at 8 kHz since no rate change is performed by this function. Its source code is found in file `iir-irs.c` and

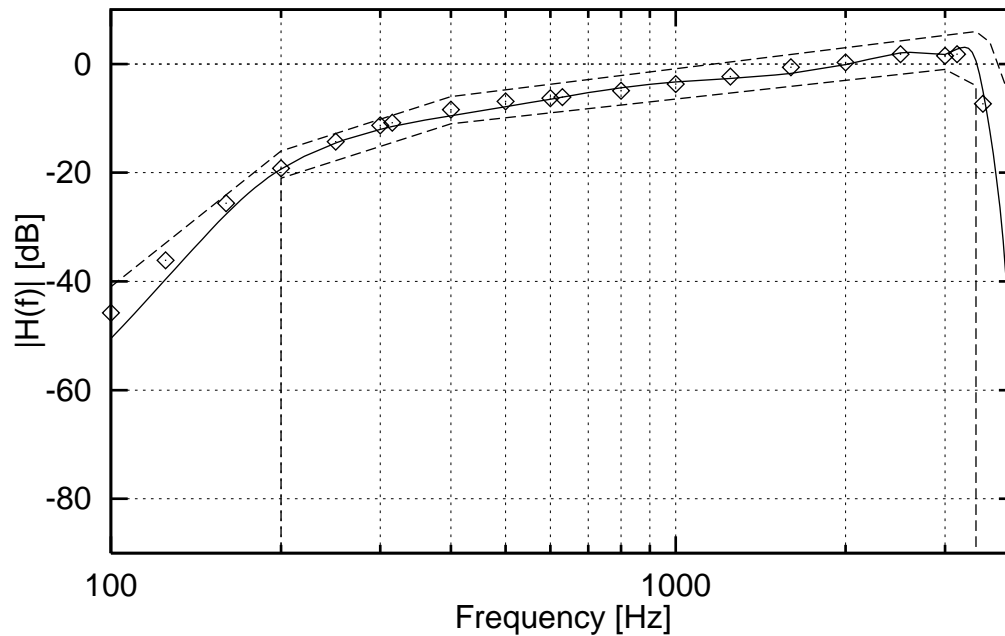


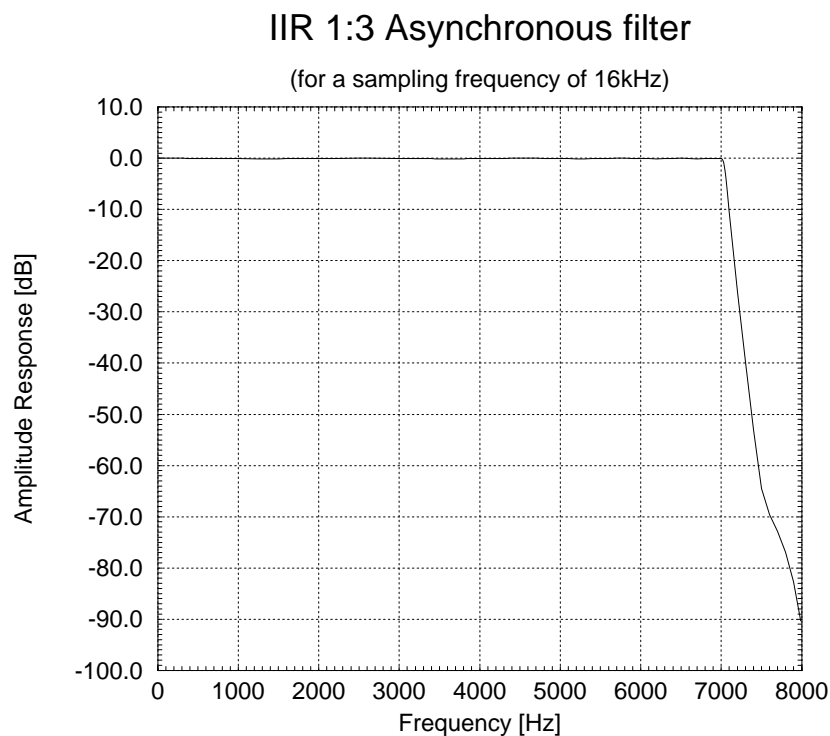
Figure 3.25: Frequency response of an IIR cascade implementation of the P.48 “full” transmit-side IRS weighting filter for data sampled at 8 kHz.

its frequency response is given in figure 3.25.

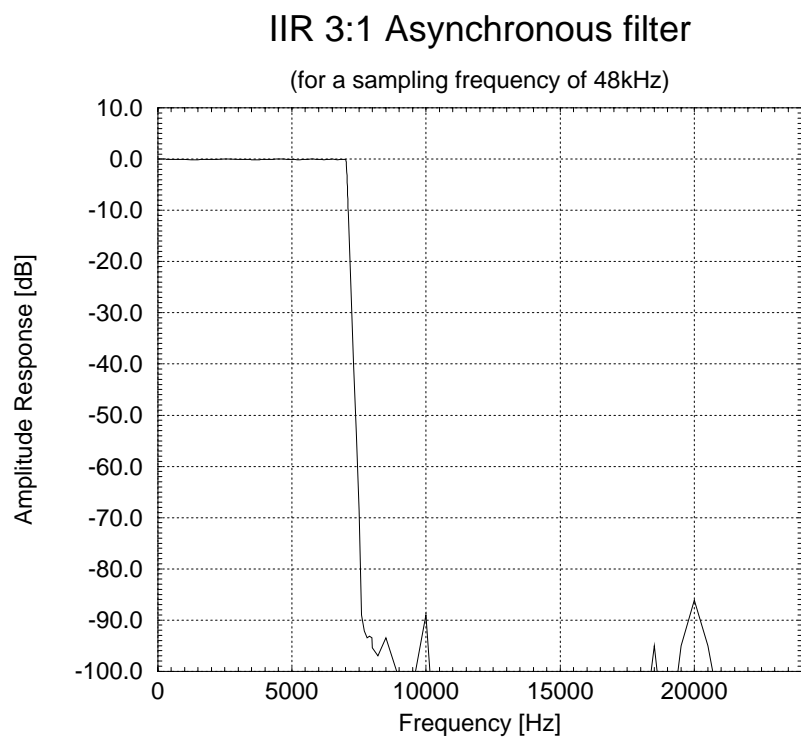
`iir_casc_lp_3_to_1_init` is the initialization routine for IIR low-pass filtering with a down-sampling factor of 3:1. Although this filter is relatively independent of the sampling rate,⁴ it was originally designed for asynchronization filtering of 16 kHz sampled speech. The -3 dB point for this filter is located at approximately 7055 Hz. Its source code is found in file `iir-flat.c` and its frequency and impulse response are given in figures 3.26(a) and 3.27(a), respectively.

`iir_casc_lp_1_to_3_init` is the initialization routine for IIR low-pass filtering with a up-sampling factor of 1:3. Although this filter is relatively independent of the sampling rate, it was originally designed for asynchronization filtering of 16 kHz sampled speech. The -3 dB point for this filter is located at approximately 7055 Hz. Its source code is found in file `iir-flat.c` and its frequency and impulse response are given in figures 3.26(b) and 3.27(b), respectively.

⁴Since this is a low-pass filter, change of sampling rate implies in change of the lower and upper cutoff frequencies.

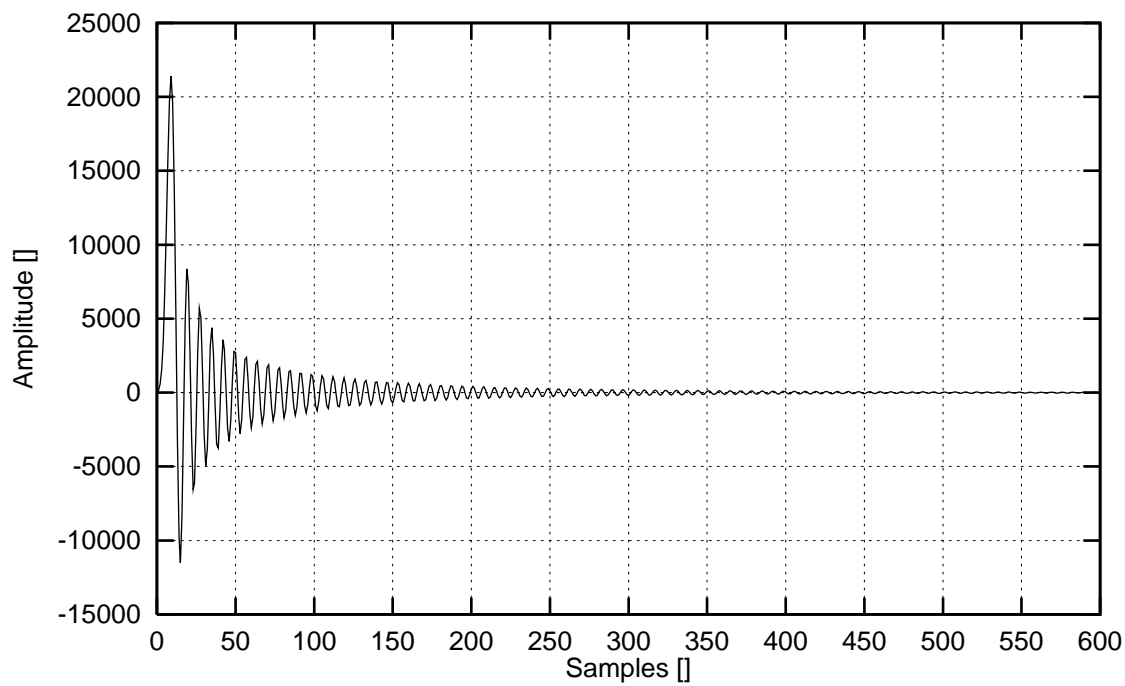


(a) Flat low-pass up-sampling by a factor of 3:1.

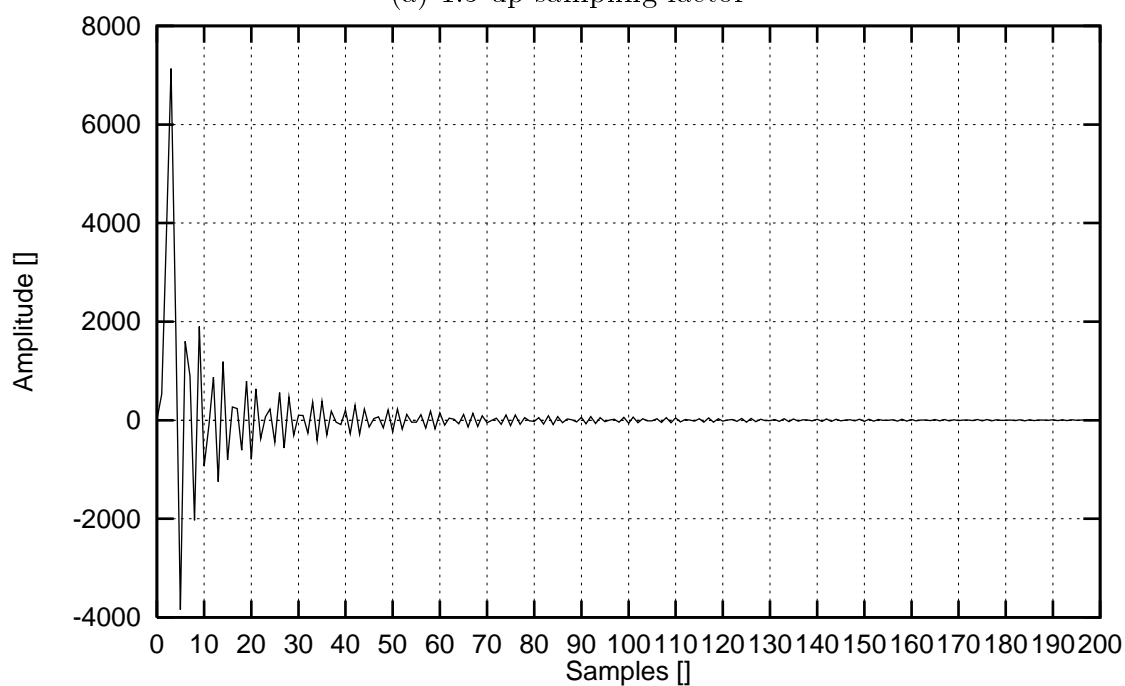


(b) Flat low-pass down-sampling by a factor of 1:3.

Figure 3.26: Flat low-pass IIR filter frequency response with factors 1:3 and 3:1 for sampling rates of 16000 and 48000 Hz.

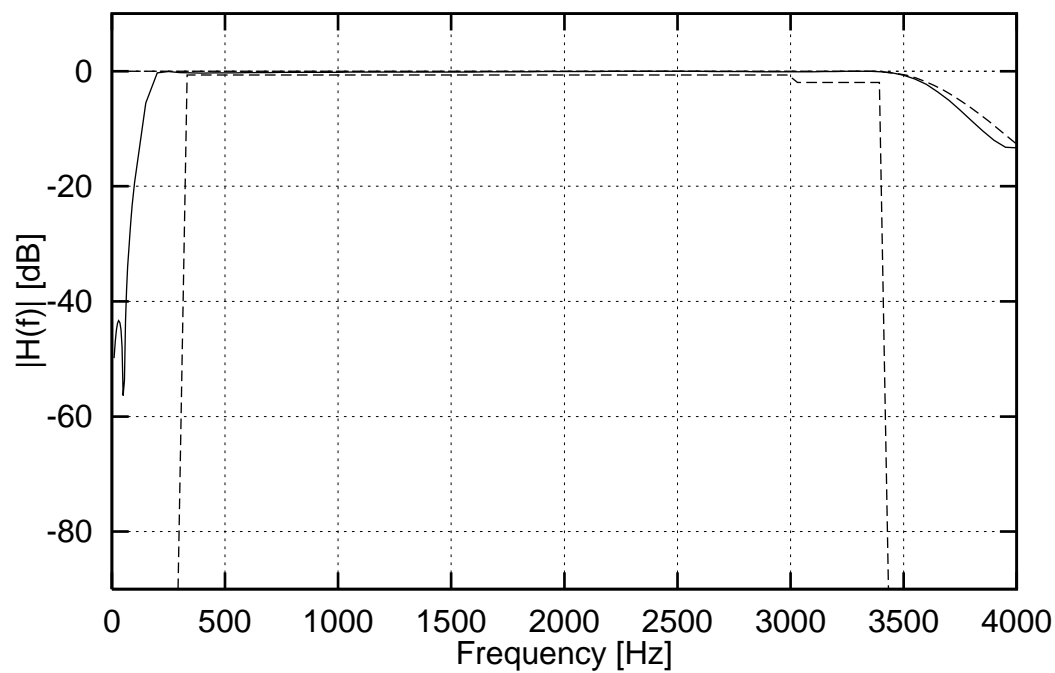


(a) 1:3 up-sampling factor

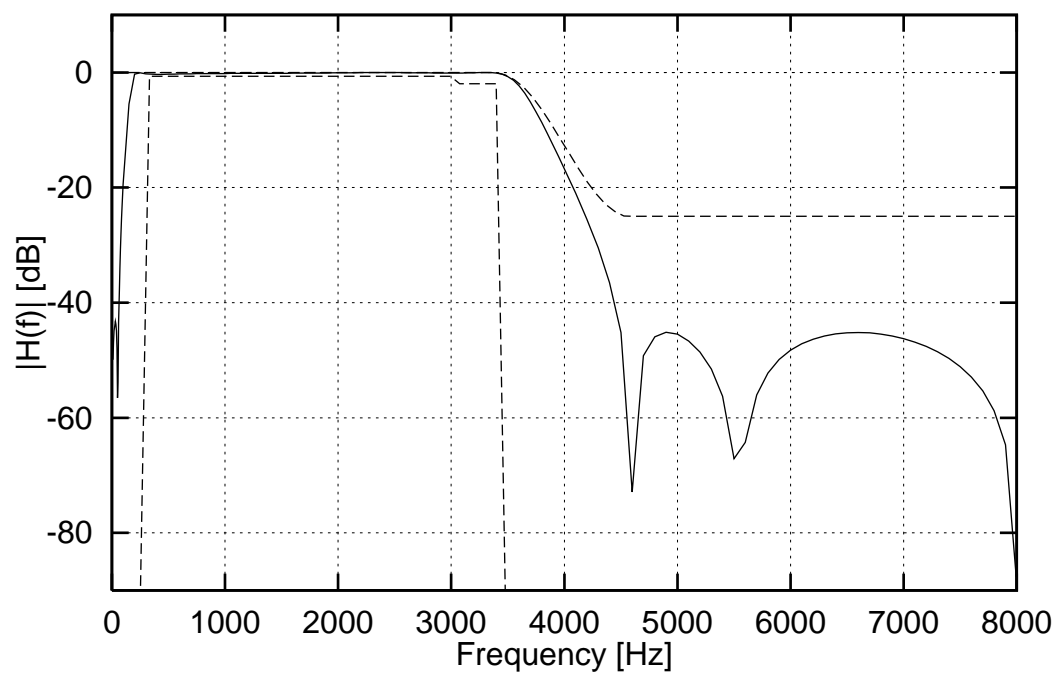


(b) 3:1 down-sampling factor

Figure 3.27: Impulse response for 1:3 and 3:1 cascade-form low-pass IIR filter.



(a) G.712 for input samples at 8 kHz, up-sampling factor 1:2



(b) G.712 for input samples at 16 kHz, down-sampling factor 2:1 or 1:1

Figure 3.28: Standard PCM (G.712) quality filter response.

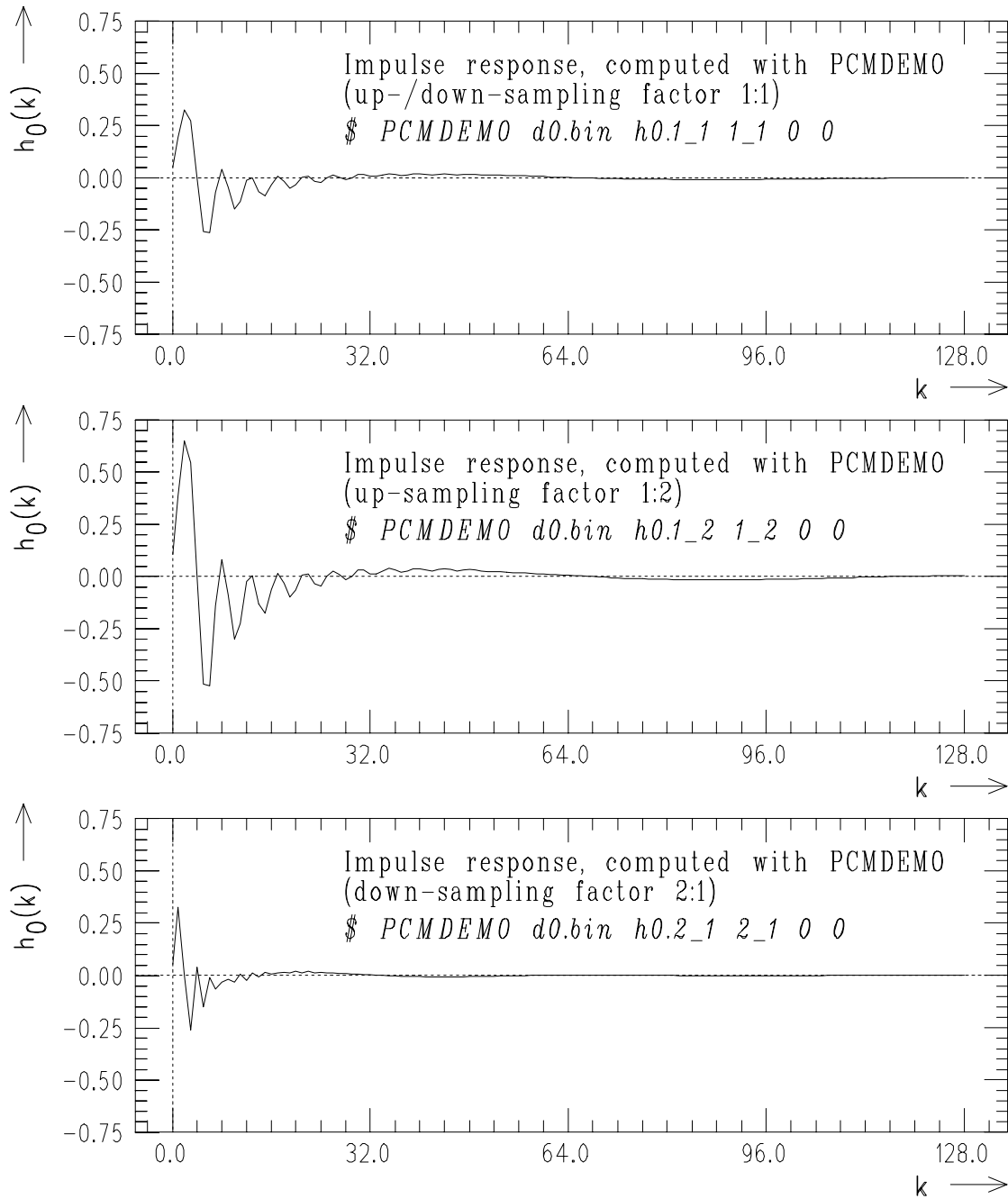


Figure 3.29: Impulse response for G.712 filters (Top: factor 1:1; Middle: factor 1:2; Bottom: factor 2:1).

3.2.2.2 cascade_iir_kernel

Syntax:

```
#include "iirflt.h"
long cascade_iir_kernel (long lseg, float *x_ptr, CASCADE_IIR *iir_ptr,
                        float *y_ptr);
```

Prototype: iirflt.h

Source code: iir-lib.c

Description:

General function for implementing filtering using a cascade-form IIR filter previously initialized by one of the `iir*_init()` routines.

Variables:

<i>lseg</i>	Number of input samples.
<i>x_ptr</i>	Array with input samples.
<i>iir_ptr</i>	Pointer to a cascade-form IIR-struct <code>CASCADE_IIR</code> .
<i>y_ptr</i>	Pointer to output samples.

Return value:

The number of output samples is returned as a `long`.

3.2.2.3 cascade_iir_reset

Syntax:

```
#include "iirflt.h"
void cascade_iir_reset (CASCADE_IIR *iir_ptr);
```

Prototype: iirflt.h

Source code: iir-lib.c

Description:

Clear state variables in `CASCADE_IIR` structure, which have been initialized by a previous call to one of the initialisation functions. Memory previously allocated is not released.

Variables:

<i>iir_ptr</i>	Pointer to struct <code>CASCADE_IIR</code> , previously initialized by a call to one of the initialization routines.
----------------	-------	--

Return value:

None.

3.2.2.4 cascade_iir_free

Syntax:

```
#include "iirflt.h"
void cascade_iir_free (SCD_IIR *iir_ptr);
```

Prototype: iirflt.h

Source code: iir-lib.c

Description:

Deallocate memory, which was allocated by an earlier call to one of the cascade-form IIR filter initialization routines described before. `iir_ptr` must not be a NULL pointer.

Variables:

`iir_ptr` Pointer to struct `CASCADE_IIR`, previously initialized by a call to one of the initialization routines.

Return value:

None.

3.2.2.5 stdpcm*_init

Syntax:

```
#include "iirflt.h"
SCD_IIR *stdpcm_16khz_init (void);
SCD_IIR *stdpcm_1_to_2_init (void);
SCD_IIR *stdpcm_2_to_1_init (void);
```

Prototypes: iirflt.h

Description:

`stdpcm_16khz_init` initializes a 16 kHz IIR filter structure for standard PCM (G712) filtering. Input and output signals will be at 16 kHz since no rate change is performed by this function. The -3 dB points for this filter are located at approximately 174 and 3630 Hz. Source code is found in file `iir-g712.c` and its frequency and impulse response are given in figures 3.28(b) and 3.29 (top), respectively.

`stdpcm_1_to_2_init` initializes standard PCM filter coefficients for filtering by the generic filtering routine `stdpcm_kernel`, for input signals at 8 kHz, generating the output at 16 kHz. The -3 dB points for this filter are located at approximately 174 and 3630 Hz. Source code is found in file `iir-g712.c` and its frequency and impulse response are given in figures 3.28(a) and 3.29 (middle), respectively.

`stdpcm_2_to_1_init` initializes standard PCM filter coefficients for filtering by the generic filtering routine `stdpcm_kernel` for input signals at 16 kHz, generating the output at 8 kHz. The -3 dB points for this filter are located at approximately 174 and 3630 Hz. Source code is found in file `iir-g712.c` and its frequency and impulse response are given

in figures 3.28(b) and 3.29 (bottom), respectively.

Variables:

None.

Return value:

This function returns a pointer to a state variable structure of type `SCD_IIR`.

3.2.2.6 `stdpcm_kernel`

Syntax:

```
#include "iirflt.h"
long stdpcm_kernel (long lseg, float *x_ptr, SCD_IIR *iir_ptr,
                   float *y_ptr);
```

Prototype: `iirflt.h`

Source code: `iir-lib.c`

Description:

General function to perform filtering using a parallel-form IIR filter previously initialized by one of the appropriate parallel-form `*_init()` routines available.

Variables:

<i>lseg</i>	Number of input samples.
<i>x_ptr</i>	Array with input samples.
<i>iir_ptr</i>	Pointer to a parallel-form IIR-struct <code>SCD_IIR</code> .
<i>y_ptr</i>	Pointer to output samples.

Return value:

This function returns the number of output samples as a `long`.

3.2.2.7 `stdpcm_reset`

Syntax:

```
#include "iirflt.h"
void stdpcm_reset (SCD_IIR *iir_ptr);
```

Prototype: `iirflt.h`

Source code: `iir-lib.c`

Description:

Clear state variables in `SCD_IIR` structure, which have been initialized by a previous call to one of the init functions. Memory previously allocated is not released.

Variables:

<i>iir_ptr</i>	Pointer to struct <code>SCD_IIR</code> , previously initialized by a call to one of the initialization routines.
----------------	-------	--

Return value:

None.

3.2.2.8 stdpcm_free**Syntax:**

```
#include "iirflt.h"
void stdpcm_free (SCD_IIR *iir_ptr);
```

Prototype: iirflt.h**Source code:** iir-lib.c**Description:**

Release memory which was allocated by an earlier call to one of the parallel-form IIR filter initialization routines described before. The parameter `iir_ptr` must not be a null pointer.

Variables:

iir_ptr Pointer to struct `SCD_IIR`, previously initialized by a call to one of the initialization routines.

Return value:

None.

3.3 Tests and portability

Compliance with the R&Os was verified by checking the frequency response of the filters and the size of the output files. Frequency response was obtained by feeding the filtering routines with sinewaves and calculating the ratio in dB, for each frequency of interest.

Portability of this module was checked by running the same speech file on a proven platform and on a test platform. Comparison of both processed files should show either no differences or yield equivalent results.⁵ Tests were performed in the VAX/VMS environment with VAX-C and gcc, in MSDOS with Borland Turbo C++ Version 1.00 and gcc (DJGPP), in SunOS with cc, acc, and gcc, and in HPUX with gcc.

⁵Some differences may appear in the output files, but for a few samples and by no more than 1 LSb. As an example, in the tests for checking VAX and SUN-OS, one of the files differed in 3 samples out of 49152 for a cascade of high-quality up- and down-sampling of 1:6 and 6:1. For small rate change factors, differences are unlikely.

3.4 Examples

3.4.1 Description of the demonstration programs

Three programs are provided as demonstration programs for the RATE module, `firdemo.c`, `iirdemo.c`, and `filter.c`.

Programs `firdemo.c` and `iirdemo.c` were the first demonstration programs for the rate change module. The former is found in directory `fir` of the STL and contains a cascade processing of the FIR filters available upto the STL96. The latter is found in directory `iir` of the STL and contains a cascade processing of the IIR filters available upto the STL96. However, because of the increasing static memory requirement for cascade processing that came with the introduction of new filters in the STL, these two programs became prohibitive and their maintenance was discontinued. They are still functional, although outdated.

Program `filter.c` is a single demonstration program that incorporates both IIR and FIR filters in the STL and has been kept up-to-date as new filters are added to the STL. Compared to the `firdemo.c` and `iirdemo.c` programs, `filter.c` can only perform one filtering operation per pass, while `firdemo.c` and `iirdemo.c` could perform a number of 1:1 operations combined with two up-sampling and two downsampling operations. Hence, several calls of the filter program are necessary to implement what was accomplished by a single call of `firdemo.c` and `iirdemo.c`, in addition to the cumulative quantization noise (from the successive float-to-short conversions). In applications where multiple filtering is needed and the user is concerned with the quantization noise accumulation, a custom-made program could be used e.g. based on a specialization of either `firdemo.c`, `iirdemo.c`, or `filter.c`.

3.4.2 Example: Calculating frequency responses

The following C code exemplifies the use of some of the filter functions available in the STL. The C code generates a number of tones which are specified by the user (lower, upper, and step frequencies). The frequency response is obtained by calculating the power change for each single frequency before and after filtered by the selected filter.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* UGST MODULES */
#include "ugstdemo.h"
#include "iirflt.h"
#include "firflt.h"

/* Other stuff */
#define TWO_PI (8*atan(1.0))
#define QUIT(m,code) {fprintf(stderr,m); exit((int)code);}
```

```

void main(argc, argv)
    int      argc;
    char     *argv[];
{
    SCD_FIR      *fir_state;
    SCD_IIR      *iir_state;
    float        *BufInp, *BufOut;
    char         F_type[20];
    long         j, N, N2;
    long         inp_size, out_size;
    double       f, f0, fstep, ff, fs, inp_pwr, H_k;
    char         is_fir;

    /* Preamble */
    N = 256; N2 = 20; inp_size = N * N2;

    /* Read parameters for processing */
    GET_PAR_S(1, "_Filter type(irs,hq2,hq3,pcm,pcm1): ... ", F_type);
    GET_PAR_D(2, "_Start frequency [Hz]: ..... ", f0);
    GET_PAR_D(3, "_Stop frequency [Hz]: ..... ", ff);
    GET_PAR_D(4, "_Frequency step [Hz]: ..... ", fstep);
    FIND_PAR_D(5, "_Sampling Frequency [Hz]: ..... ", fs, 8000);

    /* Check consistency of upper and lower frequencies */
    ff = (ff >= fs / 2)? (fs / 2) : ff;
    if (f0 < 2.0 / (double) inp_size * fs && f0 != 0.0)
        f0 = 2.0 / (double) inp_size * fs;

    /* Normalization of frequencies */
    f0 /= fs; ff /= fs; fstep /= fs;

    /* Set flag to filter type: IIR or FIR */
    is_fir = (strcmp(F_type,"pcm",3)==0 || strcmp(F_type,"PCM",3)==0)
        ? 0 : 1;

    /* ... CHOOSE CORRECT FILTER INITIALIZATION ... */
    /*
     * Filter type: irs - IRS weighting 2:1 or 1:2 factor:
     *             . fs == 8000 -> up-sample: 1:2
     *             . fs == 16000 -> down-sample: 2:1
     */
    if (strcmp(F_type, "irs", 3) == 0 || strcmp(F_type, "IRS", 3) == 0)
    {
        if (fs == 8000)
            fir_state = irs_8khz_init();
        else if (fs == 16000)
            fir_state = irs_16khz_init();
        else
            QUIT("IRS Implemented only for 8 and 16 kHz\n", 15);
    }
}

```

```

/*
 * Filter type: hq2 - High-quality 2:1 or 1:2 factor:
 *
 *         . fs == 8000 -> up-sample: 1:2
 *         . fs == 16000 -> down-sample: 2:1
 *
 *         hq3 - High-quality 3:1 or 3:1 factor
 *         . fs == 8000 -> up-sample: 1:3
 *         . fs == 16000 -> down-sample: 3:1
 */
else if (strncmp(F_type,"hq",2)==0 || strncmp(F_type,"HQ",2)==0)
{
    if (fs == 8000)                /* It is up-sampling! */
        fir_state = F_type[2] == '2'
            ? fir_up_1_to_2_init()
            : fir_up_1_to_3_init();
    else                          /* It is down-sampling! */
        fir_state = F_type[2] == '2'
            ? fir_down_2_to_1_init()
            : fir_down_3_to_1_init();
}
/*
 * Filter type: pcm - Standard PCM quality 2:1 or 1:2 factor:
 *
 *         . fs == 8000 -> up-sample: 1:2
 *         . fs == 16000 -> down-sample: 2:1
 *
 *         pcm1 - Standard PCM quality with 1:1 factor
 *         . fs == 8000 -> unimplemented
 *         . fs == 16000 -> OK, 1:1 at 16 kHz
 */
else if (strncmp(F_type,"pcm",3)==0 || strncmp(F_type,"PCM",3)==0)
{
    if (strncmp(F_type,"pcm1", 4)==0 || strncmp(F_type,"PCM1",4)==0)
    {
        if (fs == 16000)
            iir_state = stdpcm_16khz_init();
        else
            QUIT("Unimplemented: PCM w/ factor 1:1 for given fs\n", 10);
    }
    else
        iir_state = (fs == 8000)
            ? stdpcm_1_to_2_init() /* It is up-sampling! */
            : stdpcm_2_to_1_init(); /* It is down-sampling! */
}

/* Calculate Output buffer size */
if (is_fir)
    out_size = (fir_state->hswitch=='U')
        ? inp_size * fir_state->dwn_up
        : inp_size / fir_state->dwn_up;
else
    out_size = (iir_state->hswitch=='U')
        ? inp_size * iir_state->idown

```

```

        : inp_size / iir_state->idown;

/* Allocate memory for input buffer */
if ((BufInp = (float *) calloc(inp_size, sizeof(float))) == NULL)
    QUIT("Can't allocate memory for data buffer\n", 10);

/* Allocate memory for output buffer */
if ((BufOut = (float *) calloc(out_size, sizeof(float))) == NULL)
    QUIT("Can't allocate memory for data buffer\n", 10);

/* Filtering operation */
for (f = f0; f <= ff; f += fstep)
{
    /* Reset memory */
    memset(BufOut, '\0', out_size * sizeof(float));

    /* Adjust top (NORMALIZED!) frequency, if needed */
    if (fabs(f - 0.5) < 1e-8/fs) f -= (0.05*fstep);

    /* Calculate as a temporary the frequency in radians */
    inp_pwr = f * TWO_PI;

    /* Generate sine samples with peak 20000 ... */
    for (j = 0; j < inp_size; j++)
        BufInp[j] = 20000.0 * sin(inp_pwr * j);

    /* Calculate power of input signal */
    for (inp_pwr = 0, j = 0; j < inp_size; j++)
        inp_pwr += BufInp[j] * BufInp[j];

    /* Convert to dB */
    inp_pwr = 10.0 * log10(inp_pwr / (double) inp_size);

    /* Filtering the whole buffer ... */
    j = (is_fir)
        ? fir_kernel(inp_size, BufInp, fir_state, BufOut)
        : stdpcm_kernel(inp_size, BufInp, iir_state, BufOut);

    /* Compute power of output signal; discard initial 2*N samples */
    for (H_k = 0, j = 2 * N; j < out_size - 2 * N; j++)
        H_k += BufOut[j] * BufOut[j];

    /* Convert to dB */
    H_k = 10 * log10(H_k / (double) (out_size - 4 * N)) - inp_pwr;

    /* Printout of gain at the current frequency */
    printf("\nH( %4.0f ) \t = %7.3f dB\n", f * fs, H_k);
}
}

```


Chapter 4

EID: Error Insertion Device

An error insertion device (EID) is used to study the behaviour of digital transmission systems and equipments under error conditions. This requires a model for the transmission channel, and an error generation algorithm. In the most general case, burst or random bit error generators are needed. In other cases, such as when evaluating mobile and wireless systems, random and bursty frame erasures are of importance.

The EID module implements these four functionalities. The model for random and bursty bit errors, and for random frame erasure is based on a linear congruential sequence random number generator, and the bit error insertion and random frame erasure are based on a two-state channel model.

The burst frame erasure function requires a more elaborated model. For the specific application of wireless systems, a model based on Markov sequences has been developed. This is known within ITU as the Bellcore model [14, 15]. It has been used in the ITU-T 8 kbit/s speech coder selection tests, and has been incorporated in the STL.

In this chapter one finds the description of both channel models, and a description of their implementation in the EID module.

4.1 Description of the Algorithm

4.1.1 Simple Channel Model

The bit error insertion algorithm of the EID is based on a channel model where (binary) data bitstreams are to be transmitted, and is based on the discrete *Gilbert Elliott channel* (GEC) model, described in [16].

This model (see figure 4.1) has two states, Good (G) and Bad or Burst (B). Associated with these two states, there are four parameters (probabilities): two relating to the probability of remaining in state G or B , and two relating to the probability of transition from the current state to the other state (i.e., the occurrence of a binary digit transition, or error).

The probabilities associated with the channel states are P and Q , P being the probability

of transition from state G to B , and Q the probability of transition from B to G . Hence, the probability of remaining in the same state is $(1 - P)$ and $(1 - Q)$ for states G and B , respectively. For a given state, there are probabilities that a change in a bit occur, and this is P_G for state G , and P_B for state B .

Therefore, the channel may be either in the good state G , where the mean bit error probability P_G is very low ($P_G \approx 0$), or in the bad state B , where the mean bit error probability P_B is rather high ($P_B \approx 0.5$).

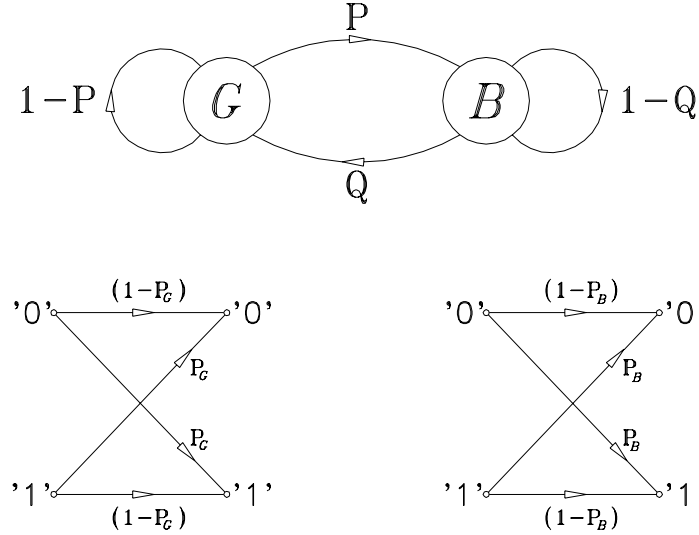


Figure 4.1: Gilbert Elliot Channel Model (GEC)

The mean bit error probability BER generated by this channel model is

$$BER = \frac{P}{1 - \gamma} \cdot P_B + \frac{Q}{1 - \gamma} \cdot P_G \quad (4.1)$$

where

$$\gamma = 1 - (P + Q) \quad (4.2)$$

is a measure for the correlation of the bit errors, and consequently an indication of the burst or random characteristic of the channel. In this issue, $\gamma \approx 0$ implies a nearly random error channel, while $\gamma \approx 1$ implies a totally bursty channel. Please note that BER is reasonable only in the range $0 \leq BER \leq 0.5$.

For many applications, bit error sequences with a distinct mean bit error probability BER and a distinct bit error correlation γ are of interest. From equations (4.1) and (4.2) we get for the remaining parameters of the GEC for arbitrarily chosen values of $0 \leq P_G < P_B \leq 0.5$:

$$P = (1 - \gamma) \cdot \left(1 - \frac{P_B - BER}{P_B - P_G}\right) \quad (4.3)$$

$$Q = (1 - \gamma) \cdot \frac{P_B - BER}{P_B - P_G} \quad (4.4)$$

In the Error Insertion Device (EID) the special values $P_G = 0$ and $P_B = 0.5$ are chosen. This relates to the fact that in the good state no bit changes are expected, hence $P_G = 0$; now, for the bad state, the channel is supposed to be in a totally uncertain state, then $P_B = 0.5$. With this choice, equations (4.3) and (4.4) reduce to:

$$P = 2 \cdot (1 - \gamma) \cdot BER \quad (4.5)$$

$$Q = (1 - \gamma) \cdot (1 - 2 \cdot BER) \quad (4.6)$$

As an example, figure 4.2 shows the effect of γ on the auto-correlation of a bitstream, generated by the *GEC* (with $BER = 0.02$ in equations (4.5),(4.6)).

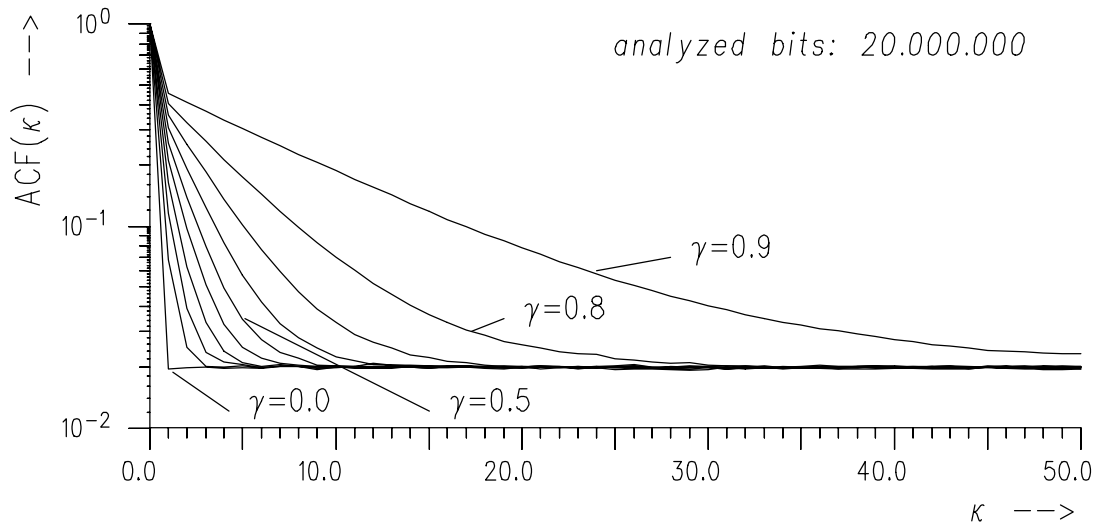


Figure 4.2: Bit Error Correlation for a Bit Error Rate of 2%

It can be seen that for $\gamma = 0$ the bit errors are statistically independent, because the autocorrelation sequence $ACF(\kappa)$ has a peak (1.0) in $\kappa = 0$, and the remaining coefficients $ACF(1), ACF(2), \dots$ oscillate around the selected bit error rate of 0.02 ($2 \cdot 10^{-2}$). For $\gamma = 0.5$, slightly bursty errors can be observed, when the initial terms of the correlation sequence build a transition region, and the remaining (higher) terms are around 0.02. Increasing γ towards unity, the correlation between the bit errors also increases, leading to totally bursty errors in the limit.

4.1.2 The Bellcore Model

The following description has been based on [14]. The actual error sequence in a wireless environment will depend upon the carrier frequency, user speed, detection scheme, type of diversity employed, mean SNR, hand-off mechanism, etc. Though a model could be

created using the above parameters, it would be impossible to apply because of the wide variance of the model parameters. It was found that a speech coder can be tested using error bursts generated by a much simpler model because the burst error performance of a speech coder can be characterized to a great extent by the way it reacts to short (5–20 ms), medium (30–60 ms) and long (over 80 ms) error bursts. If a coder performs well in the presence of a representative range of short, medium and long error bursts, it can be expected that the coder will behave well in an actual wireless communications environment, even though the actual radio channel generates error bursts with different statistics.

The Bellcore model, rather than modeling the wireless communications channel, models the occurrence of these short, medium, and long error bursts that would enable the characterization of the coder reaction to error bursts and to the error burst patterns it is expected it will encounter in practice.

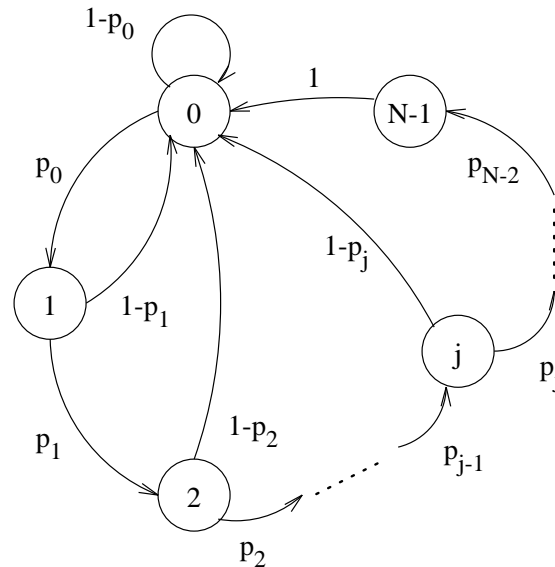


Figure 4.3: N-state Markov Model.

An N-state Markov model, as illustrated in figure 4.3, is used in the STL to generate frame erasure bursts. This model has to be adequate to test speech coders using short speech segments (6–8s). In this model, a transition from any state (0..N–1) to state 0 represents a frame received without errors, while a transition from state $j-1$ to state j indicates that j previous frames have been received in error. A transition from j back to 0 marks the end of an error burst of length j followed by a good frame.

The Markov model with N states for creating a bursty wireless communications channel is capable of erasing up to $N-1$ frames. The model generates both frames with correlated frame erasures and error-free frames. The value of N will depend on the frame duration of the speech coder under test and the maximum error burst length the coder expects to find in practice. The error statistics can be controlled by selecting the $N-1$ transition probabilities p_k , $k=0..N-2$. The probabilities will also determine the sequence of good and erased frames.

The steady state probabilities can be calculated by solving the state transition matrix or using numerical methods. If S_j denotes the steady state probability that the chain is in state j and p_j is the probability of transitioning from state j to $j + 1$, the following relationships can be established:

$$S_{j+1} = p_j S_j \quad 0 \leq j \leq N - 2$$

$$S_0 = \sum_{j=0}^{N-1} S_j (1 - p_j) \quad (p_{N-1} = 0)$$

The equations above can be solved since the probabilities should satisfy:

$$\sum_{j=0}^{N-1} p_j = 1$$

A frame erasure length of j can occur only if the chain first enters state j , and then transitions to state 0. The probability P_{fe} of this occur is

$$P_{fe} = S_j (1 - p_j)$$

The probability of receiving a frame in error can be calculated as

$$P_e = \sum_{j=1}^{N-1} j P_{fe}(j)$$

and the probability of receiving an error-free frame is $1 - P_e$. It can be seen that the steady state probability of being in state 0, S_0 , also gives the probability of receiving a frame without error, i.e.,

$$S_0 = 1 - \sum_{j=1}^{N-1} j P_{fe}(j)$$

The frame error distribution can be controlled by selecting the transition probabilities.

4.2 Implementation

The EID algorithm is made in C can be found in the module `eid.c`, with prototypes in `eid.h`. This version evolved from previous C implementations developed by PKI¹, and was used in the Host Laboratory Sessions of ETSI's contest for the second generation of the GSM Digital Mobile Radio Systems, and in the Selection Phase of the ITU-T 8 kbit/s speech coder.

The random-number generator is based on a linear congruential technique, as described in [17]. The rule here is:

$$a_n = (69069 * a_{n-1} + 1) \bmod 2^{32},$$

¹Phillips Communications Industry.

which is converted (mapped) to a float number between 0 and 1.

Since the random number generator and the channel need their internal state to be saved, two state variable data structure types were defined for the EID module. The structure type called **SCD_EID** is applicable to the burst and random bit erasure functions, as well as to the random frame erasure function. The fields of this structure are:

<i>seed</i>	Seed for random number generator.
<i>nstates</i>	Number of states of the channel model (presently 2).
<i>current_state</i>	Index of current channel state.
<i>ber</i>	Pointer to array containing thresholds according to the bit error rate in each state.
<i>usrber</i>	User defined bit error rate.
<i>usrgamma</i>	User defined correlation factor.
<i>matrix</i>	Pointer to matrix containing thresholds according to the probabilities for changing from one state to another one.

For burst frame erasures only, a different state variable structure type called **BURST_EID** has been defined, whose fields are:

<i>seedptr</i>	Memory for random number generator.
<i>internal</i>	Array with probabilities for each state of the Markov process.
<i>index</i>	Channel's current state.

The values of the fields shall not be altered and are not needed by user.

The random number generator always starts from the same point, if the user does not specify different initial seeds. In order to avoid this, the EID state variables should be saved at the end of the processing of a speech sequence, e.g. to a file by the user. This saving is not implemented by the EID module because this involves I/O to the computer file systems, and this would violate one of the UGST guidelines. Nevertheless, an example of this procedure is described in the demonstration programs that accompany this release of the EID. Therefore, users should keep in mind that, unless they save (e.g. to a file) the EID state at the end of the processing, identical error patterns will be produced, when the processing is re-started.

The sample buffers used by the EID module use *softbits*. Softbits are defined as a multi-level representation of the binary ("hard") bits '1' and '0' which are associated to probabilities of being in error. The softbit definition adopted in the ITU-T STL uses 16-bit words as representation of the hardbits '1' and '0', where the hardbit '1' is represented by the softbit 0x0081 and the hardbit '0' is represented by the softbit 0x007F. Therefore, there are 8 significant bits for each softbit; this definition is flexible enough to accomodate all applications that utilize the softbit concept. When soft-decision is not used, the hard bit information can be derived directly from bit 7 of the 16-bit softbit word. Also, the softbit '1' is represented by the softbit 0x0081, instead of 0x0080, in order to have both

hard bits '1' and '0' equally spaced from 0x0000 (in other words, 0x0000 is exactly the middle of the two-complement range for 0x81 and 0x7F). Therefore, a softbit 0x0000 represents a *total uncertainty* about the true bit value.

Error patterns produced and used by the EID module use this softbit definition. Input and output data (i.e. signals which are affected by bit errors or frame erasures) also use softbits, but additionally have a so-called synchronization header.

The *synchronization header* is defined as two consecutive 16-bit words, the first one always being a synchronization (sync) word in the range 0x6B21 to 0x6B2F, followed by the bitstream length word, a two-complement number indicating the number of softbits in the frame. The sync word 0x6B20 is reserved to indicate that a frame erasure happened. For example, for the RPE-LTP algorithm, which uses 260 bits per frame, the soft bitstream would have the format indicated in figure 4.4. It can be seen that each RPE-LTP frame will have 262 16-bit words, being one for the sync word (0x6B21 in the example), one for the frame length word (whose value here is 260), followed by 260 softbit words (here corresponding to '1', '0', ..., '0', '0'). This combination of the synchronization header and a softbit "payload" is called the bitstream signal representation and is used in ITU-T Recommendation G.192 [18] to represent encoded signals between speech encoders, error-insertion devices, transmission channel models and speech decoders.

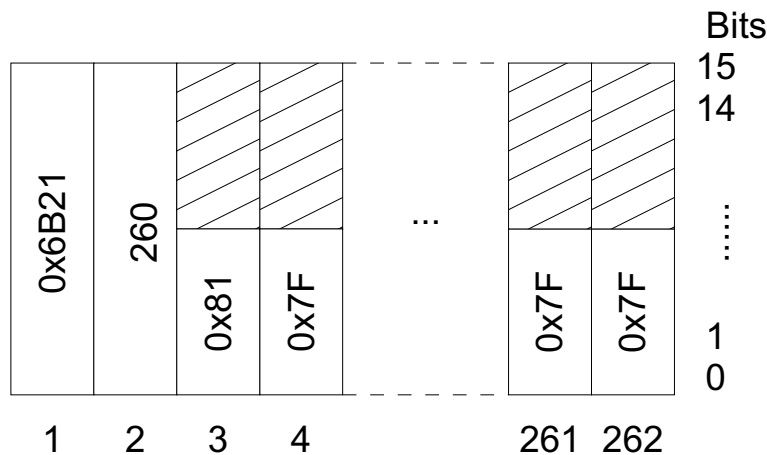


Figure 4.4: Soft bitstream format for the 13 kbit/s RPE-LTP algorithm, where 260 bits are transmitted per 20 ms transmission frame.

The EID routines for random bit errors are `BER_generator` and `BER_insertion`; for random frame erasures, `FER_generator_random` and `FER_module`; for burst frame erasures `FER_generator_burst`; and `open_eid`, `open_burst_eid` and `close_eid` for initialization (allocation) and release of EID state structures `SCD_EID` and `BURST_EID`. Their description can be found next. Besides these, there are other routines which are local (private) to the EID module, and therefore are not described.

4.2.1 open_eid

Syntax:

```
#include "eid.h"
SCD_EID *open_eid (double ber, double gamma);
```

Prototype: eid.h

Description:

Allocate memory for EID struct, set up the transmission matrix according to the selected bit error rate, and initialize the seed for the random number generator. If the symbol `PORT_TEST` is defined at compilation time, then the seed will always be initialized to the same value; otherwise, the seed is initialized with the system time (in seconds). The former is used to test portability of the EID module, since identical patterns will be generated².

Variables:

ber User desired bit error rate;
gamma User desired burst factor;

Return value:

Returns a pointer to struct `SCD_EID`; if the initialization failed, returns a null pointer.

4.2.2 open_burst_eid

Syntax:

```
#include "eid.h"
BURST_EID *open_burst_eid (long index);
```

Prototype: eid.h

Description:

Allocate memory for a state variable structure of type `BURST_EID` and setup the transmission matrix according the burst frame erasure rate (BFER) selected by *index*, and initialize the seed for the random number generator. If the symbol `PORT_TEST` is defined at compilation time, then the seed will always be initialized to the same value; otherwise, the seed is initialized with the system time, in seconds (see note in the description of `open_eid()`).

Variables:

index Indicates one of the three BFER defined for the Bellcore model. If *index* is equal to 0, the 1% BFER is selected; if 1, BFER of 3% is used; or, if *index* equals 2, a 5% BFER is used;

²Another way to force the EID to produce identical bit error patterns is to save the EID state variable (of type `SCD_EID`) e.g. to a file and, in the next call to the routine, initialize the state variable with the saved value.

Return value:

This function returns a pointer to a structure of type BURST_EID. If the initialization failed, it returns a null pointer.

4.2.3 reset_burst_eid**Syntax:**

```
#include "eid.h"
void reset_burst_eid (BURST_EID *burst_eid);
```

Prototype: eid.h

Description:

Reset a BURST_EID structure previously initialized by a call to open_burst_eid(). By default, only counters are reset; if the symbol RESET_SEED_AS_WELL is defined at compilation time, the seed is also reset. However, this is not recommended.

Variables:

burst_eid BURST_EID structure to be reset.

Return value:

None.

4.2.4 close_eid**Syntax:**

```
#include "eid.h"
void close_eid (SCD_EID *EID);
```

Prototype: eid.h

Description:

Release the memory previously allocated by open_eid() for the specified EID structure.

Variables:

EID EID state variables' structure to be released.

Return value:

None.

4.2.5 BER_generator**Syntax:**

```
#include "eid.h"
double BER_generator (SCD_EID *EID, long lseg, short *EPbuff);
```

Prototype: eid.h

Description:

Generates a softbit error pattern according to the selected channel model present in *EID*. The introduction of the bit errors in the bitstream is done by the function `BER_insertion`. It should be noted that softbit error pattern buffers do not contain synchronization headers.

Variables:

EID Structure with channel model.
lseg Length of current frame.
EPbuff Bit error pattern buffer with softbits.

Return value:

The bit error rate in the current frame is returned as a `double`.

4.2.6 FER_generator_random**Syntax:**

```
#include "eid.h"
double FER_generator_random (SCD_EID *EID);
```

Prototype: eid.h

Description:

Decides whether a random frame erasure should happen for the current frame according to the state of the GEC model in the channel memory pointed by *EID*.

Variables:

EID Structure with channel model.

Return value:

Returns a `double` value: 0 if the current frame should not be erased (“good frame”) and 1 if the frame should be erased (“bad frame”).

4.2.7 FER_generator_burst**Syntax:**

```
#include "eid.h"
double BER_generator_burst (BURST_EID *EID);
```

Prototype: eid.h

Description:

Decides whether a burst frame erasure should happen for the current frame according to the state of the Bellcore model in the channel memory pointed by *EID*. It should be noted that in the long run, the overall burst frame erasure rate (BFER) may not be consistent with the BFER specified by the user (1%, 3%, or 5%). This is an inherent deficiency of the implemented model and the calling program is responsible for computing the overall

BFER and monitoring whether this overall BFER is close enough to the desired BFER.

Variables:

EID Structure with Bellcore model parameters.

Return value:

This function returns a `double` value: 0 if the current frame should not be erased (“good frame”) and 1 if the frame should be erased (“bad frame”).

4.2.8 BER_insertion

Syntax:

```
#include "eid.h"
void BER_insertion (long lseg, short *xbuff, short *ybuff, short
                  *error_pattern);
```

Prototype: eid.h

Description:

Disturbs an input bitstream *xbuff* according to the error pattern provided in *error_pattern*, saving the disturbed bitstream in the output buffer *ybuff*. The input and output bitstream are compliant to the bitstream format described before, i.e. are comprised of a synchronization header (sync word followed by a frame length word) and softbits representing the encoded bitstream. The sync and frame length words are always located in the offsets 0 and 1 of the array, respectively. The error pattern contains only softbits. The following summarizes the bit error insertion rules:

a) input signal (after synchronization header):

- hard bit '0' represented as 0x007F;
- hard bit '1' represented as 0x0081.

b) error pattern:

- the probability for undisturbed transmission has values in the range 0x0001..-0x007F, being 0x0001 the lowest probability.
- the probability for disturbed transmission has values in the range 0x00FF..-0x0081, being 0x00FF the lowest probability.

c) output signal computation (does not affect the synchronization header, which is copied unmodified from the input buffer to the output buffer):

For input '1' (0x0081):

- if the error pattern is in the range 0x00FF..0x0081 (255..129), then the output will be 0x0001..0x007F (1..127), respectively;
- if the error pattern is in the range 0x0001..0x007F (1..127), then the output will be 0x00FF..0x0081 (255..129), respectively.

For input '0' (0x007F):

- if the error pattern is in the range 0x00FF..0x0081 (255..129), then the output will be 0x00FF..0x0081 (255..129), respectively;
- if the error pattern is in the range 0x0001..0x007F (1..127), then the output will be 0x0001..0x007F (1..127), respectively.

Variables:

lseg Length of current frame (including synchronisation header).
xbuff Buffer with input bitstream of length *lseg*.
ybuff Buffer with output bitstream of length *lseg*.
error_pattern Buffer with error pattern (without synchronisation header), of length *lseg*-2.

Return value:

None.

4.2.9 FER_module

Syntax:

```
#include "eid.h"
double FER_module (SCD_EID *EID, long lseg, short *xbuff, short *ybuff);
```

Prototype: eid.h

Description:

Implementation of the frame erasure function based on the GEC model allowing a variable degree of burstiness (as specified by parameter **gamma** in the state variable structure of type SCD_EID pointed by *EID*). This function actually erases the current frame (as described below), as opposed to function **FER_generator_random()**, which only indicates whether the current frame should be erased.

- computes the “frame erasure pattern”;
- erases all bits in one frame according the current state of the pattern generator.

The input (undisturbed) and output (disturbed) buffers have samples conforming to the bitstream representation description in Annex B of G.192. The input and output bitstream are compliant to the bitstream format described before, i.e. are comprised of a synchronization header (sync word followed by a frame length word) and softbits representing the encoded bitstream. The sync and frame length words are always located in the offsets 0 and 1 of the array, respectively. Should the frame be erased (depending on the frame erasure pattern), all softbits are set to 0x0000, which corresponds to a total uncertainty about the true bit values.

In addition, the lower 4 bits of the sync word in the synchronization header are set to 0. This makes it easier for the succeeding software to detect an erased frame. The frame length word is copied unmodified to the output buffer.

Variables:

EID Pointer to a state variable structure structure of type SCD_EID.

<i>lseg</i>	Length of current frame (including synchronisation header).
<i>xbuff</i>	Pointer to input bitstream. The synchronisation word (<i>xbuff</i> [0]) is processed, the frame length word (<i>xbuff</i> [1]) is not changed.
<i>ybuff</i>	Buffer with output bitstream.

Return value:

This function returns a `double` value: 1 if the current frame has been erased, and 0 otherwise.

4.3 Tests and portability

Portability may be checked by running the same speech file on a proven platform and on a test platform, for the whole range of input parameters. Results should be identical when the compilation is done with the symbol `PORT_TEST` properly defined and the channel states are set to a same value.

This routine had portability tested for VAX/VMS with VAX-C, MS-DOS with Turbo C v2.0, Sun-OS with Sun-C, and HP-UX with gcc.

4.4 Examples

4.4.1 Description of the demonstration programs

Two programs are provided as demonstration programs for the EID module, `eiddemo.c` (version 3.2), `eid8k.c` (version 3.2), `eid-xor.c` (version 1.0), `gen-patt.c` (version 1.4), `ep-stats.c` (version 2.0), and `eid-int.c` (version 1.0).

Program `eiddemo.c` uses input and output file in the form of a serial bit stream conforming to the bitstream signal representation, as defined in Annex B of ITU-T G.192. This program will disturb the input bitstream with errors using the Gilbert Elliot Channel model for random or burst bit error insertion and for random frame erasures. The Bellcore model, which is used for burst frame erasures, is supported as a command line option, but not as default. It should be noted that this program uses function `FER_module()`, not function `FER_generator_random()`, for random frame erasures.

Program `eid8k.c` was developed during the standardization process of the ITU-T G.729 8 kbit/s speech codec for the task of producing bit error masks which would be used in the host laboratory hardware-implemented EID. For this program, input files are not generated, but only bit error pattern files. Consistent with the definition in the STL, error patterns do not have synchronization headers (sync word and frame length word), but only softbits representing disturbances of the channel. GEC and the Bellcore model are supported in this program. The output file format is, as was necessary for the G.729 work, different from a serial bitstream as defined in the STL because the softbits are saved as `char` (8-bit words) rather than as `short` (16-bit words). Conversion of this format to the STL 16-bit bitstream format can be accomplished using the unsupported program `ch2sh.c`.

It should be noted that both programs save in files the current state of the EID models under use and also try to read these state files at startup time (if not found, the programs create new ones, which are updated when the programs terminate).

Program `eid-xor.c` is an error-insertion program that simply XORs bits in a bitstream file (in one out of three formats: G.192, byte-oriented G.192, and compact) with error patterns (bit errors or frame erasures in one out of three formats) and saves the disturbed bitstream in a file. The error patterns need not have been produced by any of the EID models implemented in the STL, they only have to be in one of the three input formats. Since error patterns are either bit error EPs or frame erasure EPs, simultaneous bit errors and frame erasures are not allowed by `eid-xor.c`.

The program `gen-patt.c` is used to generate error patterns (EPs) using the EID models implemented in the STL (Gilbert and Bellcore models). The EPs will be either frame erasures or bit errors EPs, since the models in the STL do not support mixed frame erasure/bit error mode.

Program `ep-stats.c` examines an error pattern file (either bit error EPs or frame erasure EPs) and displays the actual BER/FER found in the EP and the distribution of number of consecutive errored bits or erased frames.

Program `eid-int.c` interpolates a frame erasure EP such that each synchronism word found in the EP is repeated a user-specified number of times. This is useful to align the frame erasures for codecs that have frame sizes that are an integer sub-multiple of each other (e.g. 10ms codecs and 20 ms codecs). In the latter example, the master EP will be the 20ms one, and the one generated by `eid-int` would be used for the 10ms codec.

As a final note, it should be reinforced that the definition of the symbol `PORT_TEST` at compilation time **will** affect the operation of the programs as explained before. If this symbol is defined, functions `open_eid()` and `open_burst_eid()` will always start from the same seed. Therefore, the output of the programs will be the same, unless EID state files are available. When that symbol is not defined at compilation time, the programs will use the run-time library function `time()` to get the seed used in functions `open_eid()` and `open_burst_eid()`.

4.4.2 Using the bit error insertion routine

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "ugstdemo.h"
#include "eid.h"

#define OVERHEAD 2
#define LSEG 2048L                               /* Frame length is FIXED! */
#define SYNCword 0x6B21

void main(argc, argv)
    int      argc;
```

```

char          *argv[];
{
    SCD_EID      *ber_st;          /* pointer to EID-structure */
    char          ifile[128], ofile[128]; /* input/output file names */
    FILE          *ifilptr, *ofilptr;   /* input/output file pointer */
    static int     EOF_detected = 0;     /* Flag to mark END OF FILE */
    double         ber;                /* bit error rate factor */
    double         gamma;              /* burst factor */
    static double   dstbits = 0;         /* distorted bits count */
    static double   prcbits = 0;         /* processed bits count */
    short          err_pat[LSEG];        /* error pattern-buffer */
    short          inp[LSEG+OVERHEAD], out[LSEG+OVERHEAD]; /* bit-buffers */

    GET_PAR_S(1, "_File with input bitstream: ..... ", ifile);
    GET_PAR_S(2, "_File for disturbed bitstream: ..... ", ofile);
    GET_PAR_D(3, "_Bit error rate      (0.0 ... 0.50): ..... ", ber);
    GET_PAR_D(4, "_Burst factor        (0.0 ... 0.99): ..... ", gamma);

    /* Open input and output files */
    ifilptr = fopen(ifile, RB);
    ofilptr = fopen(ofile, WB);

    /* Allocate EID buffer for bit errors */
    ber_st = open_eid(ber, gamma);
    if (ber_st == (SCD_EID *) 0)
        QUIT(" Could not create EID for bit errors!\n", 1);

    /* Now process serial soft bitstream input file */
    while (fread(inp, sizeof(short), LSEG+OVERHEAD, ifilptr) == LSEG+OVERHEAD)
    {
        if (inp[0] == SYNCword && EOF_detected == 0)
        {
            /* Generate Error Pattern */
            dstbits += BER_generator(ber_st, LSEG, err_pat);

            /* Modify input bitstream according the stored error pattern */
            BER_insertion(LSEG+OVERHEAD, inp, out, err_pat);
            prcbits += (double) LSEG; /* count number of processed bits */

            /* Write disturbed bits to serial soft bitstream output file */
            fwrite(out, sizeof(short), LSEG+OVERHEAD, ofilptr);
        }
        else
            EOF_detected = 1;          /* the next SYNC-word is missed */
    }

    if (EOF_detected == 1)
        printf("   --- end of file detected (no SYNCword match) ---\n");
    printf("\n");
}

```

```

/* Print message with measured bit error rate */
if (prcbits > 0)
    printf("Measured BER: %f  (%ld of %ld bits distorted)\n",
           dstbits / prcbits, (long) dstbits, (long) prcbits);
}

```

4.4.3 Using the frame erasure routine

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "ugstdemo.h"
#include "eid.h"

#define QUIT(m,code) {fprintf(stderr,m); exit((int)code);}
#define LSEG 2048L                /* Frame length is FIXED! */
#define OVERHEAD 2
#define SYNCword 0x6B21

void main(argc, argv)
    int      argc;
    char     *argv[];
{
    SCD_EID   *FEReid;                /* pointer to EID-structure */
    char      ifile[128], ofile[128]; /* input/output file names */
    FILE      *ifilptr, *ofilptr;     /* input/output file pointer */
    static int EOF_detected = 0;      /* Flag to mark END OF FILE */
    double    fer;                    /* frame erasure factor */
    double    gamma;                  /* burst factor */
    static double ersfrms = 0;         /* total distorted frames */
    static double prcfrms = 0;         /* number of processed frames */
    short     inp[LSEG+OVERHEAD], out[LSEG+OVERHEAD]; /* bit-buffers */

    GET_PAR_S(1, "_File with input bitstream: ..... ", ifile);
    GET_PAR_S(2, "_File for disturbed bitstream: ..... ", ofile);
    GET_PAR_D(3, "_Frame erasure rate (0.0 ... 0.50): ..... ", fer);
    GET_PAR_D(4, "_Burst factor          (0.0 ... 0.99): ..... ", gamma);

    /* Open input and output files */
    ifilptr = fopen(ifile, RB);
    ofilptr = fopen(ofile, WB);

    /* Allocate EID buffer for bit errors frame erasure */
    FEReid = open_eid(fer, gamma);
    if (FEReid == (SCD_EID *) 0)
        QUIT(" Could not create EID for frame erasure module\n", 1);
}

```



```
/* Now process serial soft bitstream input file */
while (fread(inp, sizeof(short), LSEG+OVERHEAD, ifilptr) == LSEG + OVERHEAD)
{
    if (inp[0] == SYNCword && EOF_detected == 0)
    {
        /* Generate frame erasure */
        ersfrms += FER_module(FEReid, LSEG+OVERHEAD, inp, out);
        prcfrms++;          /* count number of processed frames */

        /* Write (erased) frames to serial soft bitstream output file */
        fwrite(out, sizeof(short), LSEG+OVERHEAD, ofilptr);
    }
    else
        EOF_detected = 1;      /* the next SYNC-word is missed */
}

if (EOF_detected == 1)
    printf("  --- end of file detected (no SYNCword match) ---\n");
printf("\n");

/* Print message with measured bit error rate */
if (prcfrms > 0)
    printf("measured FER: %f  (%ld of %ld frames erased)\n",
           ersfrms / prcfrms, (long) ersfrms, (long) prcfrms);
}
```


Chapter 5

G.711: The ITU-T 64 kbit/s log-PCM algorithm

In the early 1960's an interest was expressed in encoding the analog signals in telephone networks, mainly to reduce costs in switching and multiplexing equipments and to allow the integration of communication and computing, increasing the efficiency in operation and maintenance [19].

In 1972, the then CCITT published the Recommendation G.711 that constitutes the principal reference as far as transmission systems are concerned [20]. The basic principle of the algorithm is to code speech using 8 bits per sample, the input voiceband signal being sampled at 8 kHz, keeping the telephony bandwidth of 300–3400 Hz. With this combination, each voice channel requires 64 kbit/s.

5.1 Description of the algorithm

The idea behind the digitalization of the network involved a compromise: use as far as possible the existing infrastructure; this imposes a bandwidth limitation for the bitstreams of coded signals. A rate of 64 kbit/s was found to be reasonable.

If one thinks of using the most natural quantization scheme, one will choose linear quantization. But one drawback of this approach is that the signal-to-noise ratio (SNR) varies with the amplitude of the input signals: the smaller the amplitude, the smaller the SNR. And, from the quality point of view, if a signal has a wide variance, or a variance that changes with time (as in the case of speech signals), the SNR will also change, resulting in a wide-varying quality of the system.

To avoid this problem, one can use logarithmic quantization, which will result into a more uniform quantization noise. With this in mind, several studies were carried out in late 1960's to choose a good algorithm for this purpose. This led to the definition of two transmission schemes, one using the μ law compression characteristic:

$$c(x) = x_{max} \frac{\ln(1 + \mu|x|/x_{max})}{\ln(1 + \mu)} \text{sgn}(x)$$

and the other using the A law compression characteristic:

$$c(x) = \begin{cases} \frac{A|x|}{1 + \ln(A)} \operatorname{sgn}(x), & \text{for } 0 \leq \frac{|x|}{x_{\max}} \leq \frac{1}{A} \\ x_{\max} \frac{1 + \ln(A|x|/x_{\max})}{1 + \ln(A)} \operatorname{sgn}(x), & \text{for } \frac{1}{A} \leq \frac{|x|}{x_{\max}} \leq 1 \end{cases}$$

Both characteristics behave as linear for small amplitude signals (being then equivalent to a linear quantization scheme), but are truly logarithmic for large signals. In fact, for large signals the SNR is:

$$SNR_{\mu} = 6.02B + 4.77 - 20 \log_{10}(\ln(1 + \mu))$$

and

$$SNR_A = 6.02B + 4.77 - 20 \log_{10}(1 + \ln A)$$

where B is the number of bits used for quantization.

The ITU chose the values $A = 87.56$ and $\mu = 255$ for the G.711 standard, together with 8 bits per sample, what leads the latter two equations to:

$$SNR_{\mu} = 6.02B - 9.99 = 38.17dB$$

and

$$SNR_A = 6.02B - 10.1 = 38.06dB$$

The G.711 standard does not specify the law as defined above, but rather uses a good linear-piecewise approximation for 8 bit samples, which has easier implementation (in hardware), as well as other properties (see [21, p.229]).

This approximation uses bit 1 for sign (1 for positive, 0 for negative), bits 2–4 to indicate a segment, and bits 5–8 for level¹. Within each segment, the quantization is linear (4 bits, or 16 levels), having 15 segments of distinct slopes for μ law, and 13 for A law.

The A law works with signals in the range from -4096 to 4096, implying in a range of 13 bits. As for the μ law, the linear signals are accepted in the range -8159 to 8159, which is represented by 14 bits. Besides this, in the dynamic range sense, A and μ law are equivalent to 12 and 13 bit linear quantization, respectively.

One detail for the A law is that the even bits are inverted. The reason for this comes from problems observed (before the standardization of the line code HDB3) in transmission systems when long sequences of zeros happen, because small amplitudes, in A law, to be coded mostly using ‘0’ bits. With this bit-inversion, long sequences of bits ‘0’ becomes less probable, thus improving performance.

The conversion rule for A/ μ law from/to linear is described in terms of tables in G.711. A good reason for this is that there is no closed form for the compression of linear samples (although it is possible to find a closed formulae for the expansion algorithm). Hence, two implementations are possible: table look-up, and algorithmic. For in-chip (LSI)

¹Please note that the bit numbering in the G.711 is the reversal of the commonly used in computer languages, G.711’s bit 1 corresponding to common-sense’s (most significant) bit 7, and G.711’s bit 8 to the normal least significant bit 0, respectively.

implementations, the first one may be preferred, because it is simpler to implement, at the cost of a wider chip area. For other applications, such as using Digital Signal Processors (DSPs), or software implementations, table look-up would occupy too much memory, and the algorithmic solution would be preferred.

5.2 Implementation

This implementation of the G.711 can be found in the module `g711.c`, with prototypes in `g711.h`.

For the reason explained before, an algorithmic approach to the G.711 was followed. For the compression routines, first the samples are converted from two's complement to signed magnitude notation². So, a segment classification is done, and then the linear quantization of a certain number of bits of the input sample, that depends on the segment number (e.g., for A law, segment 1 uses a factor of 2:1, 2 a 4:1, etc.) is carried out. Finally, the sign of the sample is added. The expansion routines are even simpler: find the sign, get the mantissa and the exponent, and compute the linear sample.

One important point here is that, following UGST Guidelines, linear input samples must be left-justified `shorts`. With this approach, the knowledge of the 0 dB reference for the file is simplified, and the need of having to apply different normalization factors to files if they are to be coded by A or μ law is eliminated³. As an example, suppose that we want to process a speech file \mathcal{X} by the G.711 at an input level of -20 dBov for both A and μ law. Then, if the sample representation is right-justified, and a factor f brings a file's level to -20 dBov for μ law, then for A law the factor will be $2.f$, due to the difference in input signal's dynamic range of both laws (4096 and 8159, respectively). On the other hand, if the samples are left-justified, the factor is only one, and the routines will only look at the 13 or 14 most significant bits of the 16-bit word, for A and μ law, respectively. In other words, the peak value for linear and A/ μ law is the same, therefore one factor is sufficient.

Compliance tests to this code have been done using a ramp file having the full excursion of the dynamic range for each of the laws, and examining the compressed and expanded samples against the values expected in tables 1a, 1b, 2a, and 2b of Recommendation G.711 (see [20]). Another test done exploits the synchronous property of the G.711 scheme. Only samples from column 7 of G.711 tables 1 and 2 were used. These values are transparent to quantization. Hence, if the coding was done properly, output samples should match exactly the original ones.

The compression functions are `alaw_compress` and `ulaw_compress`, and the expansion functions are `alaw_expand` and `ulaw_expand`. In the next part you find a summary of calls to these functions.

²Using the samples as two's complement in the compression algorithm is a very common error whose effects are only noticeable for small amplitude signals. Our approach agrees to the one in G.726[22], block *compress*.

³In the case of stand-alone tools, this would mean that two copies of the same file should be available!

5.2.1 alaw_compress and ulaw_compress

Syntax:

```
#include "g711.h"
void alaw_compress (long smpno, short *lin_buf, short *log_buf)
void ulaw_compress (long smpno, short *lin_buf, short *log_buf)
```

Prototype: g711.h

Description:

`alaw_compress` performs A law encoding rule according to ITU-T Recommendation G.711, and `ulaw_compress` does the same for μ law. Note that input samples shall be left-justified, and that the output samples are right-justified with 8 bits.

Variables:

<i>smpno</i>	Is the number of samples in <i>lin_buf</i> .
<i>lin_buf</i>	Is the input samples' buffer; each short sample shall contain linear PCM (2's complement, 16-bit wide) samples, left-justified.
<i>log_buf</i>	Is the output samples' buffer; each short sample will contain right-justified 8-bit wide valid A or μ law samples.

Return value: None.

5.2.2 alaw_expand and ulaw_expand

Syntax:

```
#include "g711.h"
void alaw_expand (long smpno, short *log_buf, short *lin_buf)
void ulaw_expand (long smpno, short *log_buf, short *lin_buf)
```

Prototype: g711.h

Description:

`alaw_expand` performs A law decoding rule according to ITU-T Recommendation G.711, and `ulaw_expand` does the same for μ law. Note that output samples will be left-justified, and that the input samples shall be right-justified with 8 bits.

Variables:

<i>smpno</i>	Is the number of samples in <i>log_buf</i> .
<i>log_buf</i>	Is the input samples' buffer; each short sample shall contain right-justified 8-bit wide valid A or μ law samples.
<i>lin_buf</i>	Is the output samples' buffer; each short sample will contain linear PCM (2's complement, 16-bit wide) samples, left-justified.

Return value: None.

5.3 Tests and portability

Portability may be checked by running the same speech file in a proven platform and in a test platform. Files processed this way should match exactly. Source and processed reference files for portability tests are provided in the STL distribution.

These routines had portability tested for VAX/VMS with VAX-C and gcc, MS-DOS with Turbo C v2.0, HPUNIX with gcc, and Sun-OS with Sun-C.

5.4 Example code

5.4.1 Description of the demonstration program

One program is provided as demonstration program for the G.711 module, `g711demo.c`.

Program `g711demo.c` accepts input files in 16-bit linear PCM format for compression operation and produces files in the same format after the expansion operation. The compressed signal will be in 16-bit, right adjusted format, according to the logarithmic law specified by the user. Three operations are possible: linear in, linear out (*lili*) linear in, logarithmic out (*lilo*), or logarithmic in, linear out (*loli*).

5.4.2 Simple example

The following C code gives an example of companding using either the A- or μ -law functions available in the STL.

```
#include <stdio.h>
#include "ugstdemo.h"
#include "g711.h"

#define BLK_LEN 256
#define QUIT(m,code) {fprintf(stderr,m); exit((int)code);}

main(argc, argv)
    int      argc;
    char     *argv[];
{
    char     law[4];

    char     FileIn[180], FileOut[180];
    short    tmp_buf[BLK_LEN], inp_buf[BLK_LEN], out_buf[BLK_LEN];
    FILE     *Fi, *Fo;
    void     (*compress)(), (*expand)(); /* pointer to a function */

    /* Get parameters for processing */
    GET_PAR_S(1, "_Law (A,u): ..... ", law);
    GET_PAR_S(2, "_Input File: ..... ", FileIn);
```

```
GET_PAR_S(3, "_Output File: ..... ", FileOut);

/* Opening input and output LOG-PCM files */
Fi = fopen(FileIn, RB);
Fo = fopen(FileOut, WB);

/* Choose compression/expansion routines according to the law */
if (toupper(law[0])=='A')
{
    compress = alaw_compress;
    expand = alaw_expand;
}
else if (tolower(law[0])=='u')
{
    compress = ulaw_compress;
    expand = ulaw_expand;
}
else
    QUIT("Bad law chosen!\n",1);

/* File processing */
while (fread(inp_buf, BLK_LEN, sizeof(short), Fi) == BLK_LEN)
{
    /* Process input linear PCM samples in blocks of length BLK_LEN */
    compress(BLK_LEN, inp_buf, tmp_buf);

    /* Process log-PCM samples in blocks of length BLK_LEN */
    expand(BLK_LEN, tmp_buf, out_buf);

    /* Write PCM output word */
    fwrite(out_buf, BLK_LEN, BLK_LEN, sizeof(short), Fo);
}

/* Close input and output files */
fclose(Fi);
fclose(Fo);
return 0;
}
```


Chapter 6

G.726: The ITU-T ADPCM algorithm at 40, 32, 24, and 16 kbit/s

In 1982, a group was established by the then CCITT Study Group XVIII to study the standardization of a speech coding technique that could reduce the 64 kbit/s rate used in digital links, as per ITU-T Recommendation G.711 (see related Chapter), by half while maintaining the same voice quality.

After considering contributions received from several organizations, there was a general feeling that the ADPCM (*Adaptive Differential Pulse Code Modulation*) technique could provide a good quality coder. This process of finalizing an algorithm took 18 months of development and objective and subjective testings, to culminate in a ITU Recommendation, published in October, 1984, and available in the Red Book series as Recommendation G.721.

Meanwhile, problems were found with the G.721 algorithm of 1984 regarding voice-band data signals modulated using the Frequency Shift Keying (FSK) technique, and changes had to be done to the algorithm. These changes were approved in 1986 and published in the next series of Recommendations of the CCITT, the Blue Book series, superseding the Red Book version of the G.721. This is why a note in the Blue Book G.721 warns the user that the bit stream of coded speech from this version is incompatible with the old one. Also in that Study Period (1985-1988), a need for other rates was identified, and a new Recommendation, G.723, was approved to extend the bitrate to 24 and 40 kbit/s.

In the Study Period of 1989–1992, these two Recommendations have been joined into a single one, keeping full compatibility with the former ones, and adding a lower rate of 16 kbit/s. This new Recommendation was named G.726, and the former G.721 and G.723 have been replaced.

The current version of the STL includes a G.726 implementation. In the section to follow, the operation of the G.726 algorithm is described only for the 32 kbit/s bit rate. A complete description of the G.726 algorithm can be found in [22]. Other analyses of the algorithm, besides some based on the Red Book version, can be found in several studies [23, 24, 25].

Despite the change in numbering, the ITU-T ADPCM algorithm for speech coding at 32 kbit/s, the term “G.721 algorithm” has been retained for simplicity of the text, although

a more formal reference should be “*G.726 at 32 kbit/s*”.

6.1 Description of the 32 kbit/s ADPCM

The basic idea behind the G.721 coder is to code into 4-bit samples the input speech-band signals, sampled at 8 kHz and represented by the 8-bit of G.711 A or μ law samples. The decoder just implements the reverse procedure.

The ADPCM algorithm of the G.721 exploits the predictability of the speech signals. Therefore, an adaptive predictor is used to compute the difference signal $d(k)$ (based on the expanded input log-pcm sample $s(k)$), which is then quantized by an adaptive quantizer using 4 bits. These bits are sent to the decoder and then fed into an inverse quantizer. The difference signal is used to calculate the reconstructed signal, $s_r(k)$, which is compressed (A- or μ -law) and output from the decoder ($s_d(k)$).

From this description, one could ask the following:

- If only the quantized signal is transmitted, how can the decoder reconstruct the signal?
- How can one assure stability of the predictor?
- Will this bitrate reduction degrade the voice quality?

These and others have already been considered in the design of the G.721, and many blocks of the algorithm are made to assure a good behaviour. For example, one possibility in this backward approach for adaptation is to have encoder and decoder starting from the same point, which is accomplished by resetting key variables to a known state (useful for implementation verification). Leak factors have been introduced to ensure that the algorithm will always converge, independently of the initial state. To avoid instabilities, some parameters had their range limited. To provide some insight in the building blocks of the G.721 algorithm, a short description of each of them is given [22, 24].

6.1.1 PCM format conversion

The input signal $s(k)$, in either A- or μ -law format, must be converted into linear samples. This expansion is accomplished using the same algorithm in G.711 [20], but converting from signed magnitude to 14-bit two's complement samples.

6.1.2 Difference Signal Computation

This block simply calculates the difference between the (expanded) input signal and the estimated signal:

$$d(k) = s_l(k) - s_e(k)$$

6.1.3 Adaptive Quantizer

A 15-level, non-uniform adaptive quantizer is used to quantize the difference signal. Before the quantization, this signal is converted to a logarithmic representation¹ and scaled by a factor ($y(k)$), that is computed in the scale factor adaptation block (see below).

The output of this block is $I(k)$, and it is used twice; first, is the ADPCM coded (*quantized*) sample; second, is the input to the backward part of the G.721 algorithm, to provide information for quantization of the next samples. One relevant point to notice here is that the backward adaptation is done using the quantized sample. If one starts the decoder from this very point, one will find identical behaviour. That is why only the quantized samples are needed in the decoder (i.e., no side information).

6.1.4 Inverse Adaptive Quantizer

The inverse adaptive quantizer takes the signal $I(k)$ and converts it back to the linear domain, generating a quantized version of the difference signal, $d_q(k)$. This is the input to the adaptive predictor, such that the estimated signal is based on a quantized version of the difference signal, instead of on the unquantized (original) one.

6.1.5 Quantizer Scale Factor Adaptation

This block computes $y(k)$, the factor used in the adaptive quantizer and inverse quantizer for domain conversion. As input, this block needs $I(k)$, but also $a_l(k)$, the adaptation speed control parameter. The reason for the latter is that the scaling algorithm has two modes (*bimodal adaptation*), one fast, another slow. This has been done to accomodate signals that in nature produce difference signals with large fluctuations (e.g. speech) and small fluctuations (e.g. tones and voice-band data), respectively.

This block computes two scale factor (fast, $y_u(k)$, and slow, $y_l(k)$) based on $I(k)$, which combined using $a_l(k)$ produce $y(k)$.

6.1.6 Adaptation Speed Control

This block evaluates the parameter $a_l(k)$, which can be seen as a *proportion* of the speed (fast or slow) of the input signal, and is in the range $[0, 1]$. If 0, the data are considered to be *slowly* varying; if 1, they are considered to be *fast* varying.

To accomplish this, two measures of the average magnitude of $I(k)$ are computed ($d_{ms}(k)$ and $d_{ml}(k)$). These, in conjunction with delayed tone detect and transition detect flags ($t_d(k)$ and $t_r(k)$, calculated in the Tone Transition and Detector block), are used to evaluate $a_p(k)$, whose delayed version ($a_p(k-1)$) is used in the definition of $a_l(k)$, limiting the range to $[0, 1]$ ².

¹Remember that to multiply samples in the linear domain one may add in the logarithmic one. Using efficient log and exponentiation algorithms (as done here), this turns out to be very advantageous.

²This limitation delays the start of a fast to slow transition until the average magnitude of $I(k)$ remains

An analysis of $a_p(k)$ gives insight on the nature of the signal: if around the value of 2, this means that the average magnitude of $I(k)$ is changing, or that a tone has been detected, or that it is idle channel noise; on the other side, if near 0, the average magnitude of $I(k)$ remains relatively constant.

6.1.7 Adaptive Predictor and Reconstructed Signal Calculator

The adaptive predictor has as its main function to compute the signal estimate based on the quantized difference signal, $d_q(k)$. It has 6 zeroes and 2 poles, structure that covers well the kind of input signals expected for the algorithm. With these coefficients, and past values of $d_q(k)$ and $s_e(k)$, the updated value for the signal estimate $s_e(k)$ is computed.

The two sets of coefficients (one for the pole section, $a_i(k), i = 1..2$, other for the zero section, $b_i(k), i = 1..6$) are updated using a simplified gradient algorithm. At this point, since a situation in which the poles cause instability may arise, the two pole coefficients a_i have their ranges limited. In addition, if a transition from partial band signal is detected (signaled by $t_r(k)$), the predictor is reset (all coefficients are set to 0), remaining disabled until t_r comes back to zero³.

The reconstructed signal $s_r(k)$ is calculated using the signal estimate $s_e(k)$ and the quantized difference signal $d_q(k)$.

6.1.8 Tone Transition and Detector

This block is one of the changes from the Red Book version. It was added to improve algorithm performance for signals originating from FSK modems operating in the character mode. First, it checks if the signal has partial band (e.g., a tone) by looking at the predictor coefficient $a_2(k)$, that defines the signal $t_d(k)$. Second, a transition from partial band signal indicator $t_r(k)$ is set, such that predictor coefficients can be set to 0 and the quantizer can be forced into the fast mode of operation.

6.1.9 Output PCM Format Conversion

This block is unique to the decoder. Its sole function is to compress the reconstructed signal $s_r(k)$, which is in linear PCM format, using A or μ law, and is a complement of the PCM format conversion block.

6.1.10 Synchronous Coding Adjustment

This block is also unique to the decoder. It has been devised in order to prevent cumulative distortions occurring on synchronous tandem codings (ADPCM-PCM-ADPCM, etc., in purely digital connections, i.e., with no intermediate analog conversions), provided that:

constant for some time; acting so, premature transitions for pulsed input signals, such as switched carrier voiceband data, are avoided.

³Note that when this happens, the quantizer is forced into the fast mode of adaptation.

- the transmission of the ADPCM and the intermediate PCM are error-free, and
- the ADPCM and the intermediate PCM are not disturbed by digital signal processing devices.

6.1.11 Extension for linear input and output signals

An extension of the G.726 algorithm was carried out in 1994 to include, as an option, linear input and output signals. The specification for such linear interface is given in its Annex A [26].

This extension bypasses the PCM format conversion block for linear input signals, and both the Output PCM Format Conversion and the Synchronous Coding Adjustment blocks, for linear output signals. These linear versions of the input and output signals are 14-bit, 2's complement samples.

The effect of removing the PCM encoding and decoding is to decrease the coding degradation by 0.6 to 1 qdu, depending on the network configuration considered (presence or absence of a G.712 filtering).

Currently, this extension has not been incorporated in the STL.

6.2 ITU-T STL G.726 Implementation

The STL implementation of the G.726 algorithm can be found in module `g726.c`, with prototypes in `g726.h`.

Originally in Fortran (VAX Fortran-77), the source was translated by means of the public-domain code converter *f2c* [27]. This explain why the code makes extensive use of passage of parameters by reference, rather than by value, and why many functions, that could be implemented as macros (using the C pre-processor directive `#define`), are routines, and as well as all routines return `void`.

The problem of storing the state variables was solved by defining a structure containing all the necessary variables, defining a new type called `G726_state`. By means of this approach, several streams may be processed in parallel, provided that one structure is assigned (and that one call to the encoding/decoding routines is done) for each data stream (this can be advantageous for machines with support for parallel processing). The G726 state variable structure has the following fields (all are `short`, except `ylp`, which is `long`):

<i>sr0</i>	Reconstructed signal with delay 0
<i>sr1</i>	Reconstructed signal with delay 1
<i>a1r</i>	Delayed 2nd-order predictor coefficient 1
<i>a2r</i>	Delayed 2nd-order predictor coefficient 2
<i>b1r</i>	Delayed 6th-order predictor coefficient 1
<i>b2r</i>	Delayed 6th-order predictor coefficient 2
<i>b3r</i>	Delayed 6th-order predictor coefficient 3
<i>b4r</i>	Delayed 6th-order predictor coefficient 4

<i>b5r</i>	Delayed 6th-order predictor coefficient 5
<i>b6r</i>	Delayed 6th-order predictor coefficient 6
<i>dq0</i>	Quantized difference signal with delay 0
<i>dq1</i>	Quantized difference signal with delay 1
<i>dq2</i>	Quantized difference signal with delay 2
<i>dq3</i>	Quantized difference signal with delay 3
<i>dq4</i>	Quantized difference signal with delay 4
<i>dq5</i>	Quantized difference signal with delay 5
<i>dmsp</i>	Short term average of the $F(I)$ sequence
<i>dmlp</i>	Long term average of the $F(I)$ sequence
<i>apr</i>	Triggered unlimited speed control parameter
<i>yup</i>	Fast quantizer scale factor
<i>tdr</i>	Triggered tone detector
<i>pk0</i>	Sign of dq+sez with delay 0
<i>pk1</i>	Sign of dq+sez with delay 1
<i>ylp</i>	Slow quantizer scale factor

The encoding function is `G726_encode`, and the decoding function is `G726_decode`. There are 41 other routines that, grouped in individual calls inside the encoder and decoder, implement the algorithm. Therefore, none of these 41 routines are expected to be accessed by the user, and only the two main ones.

In the following part a summary of calls to both functions is found.

6.2.1 G726_encode

Syntax:

```
#include "g726.h"
void G726_encode (short *inp_buf, short *out_buf, long smpno, char
                  *law, short rate, short reset, G726_state *state)
```

Prototype: g726.h

Description:

Simulation of the ITU-T G.726 ADPCM encoder. Takes the A or μ law input array of shorts *inp_buf* (16 bit, right-justified, without sign extension) with *smpno* samples, and saves the encoded samples in the array of shorts *out_buf*, with the same number of samples and right-justified. An example of the sample packing for the G.726 encoded bitstream is shown in figure 6.1.

The state variables are saved in the structure *state*, and the reset can be established by making *reset* equal to 1. The law is A if *law*== '1', and mu law if *law*== '0'.

Variables:

<i>inp_buf</i>	Is the input samples' buffer; each short sample shall contain right-justified 8-bit wide valid A or μ law samples.
<i>out_buf</i>	Is the output samples' buffer; each short sample will contain

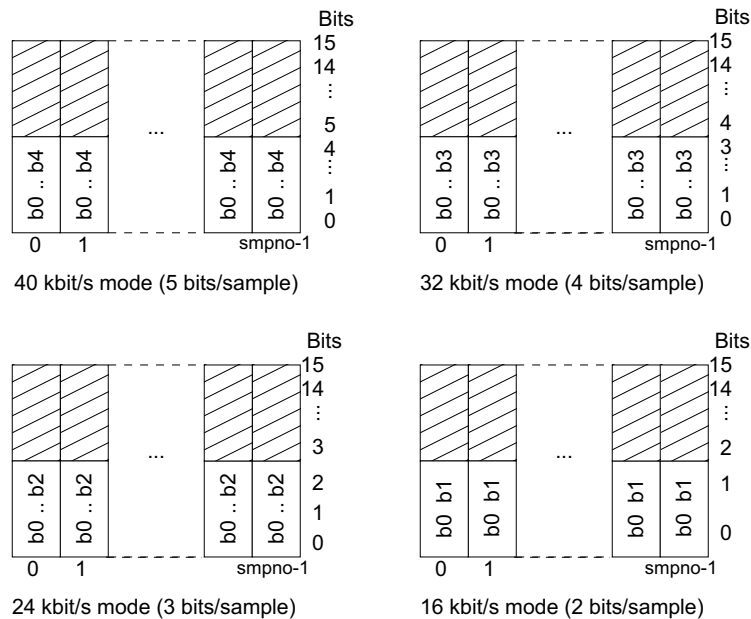


Figure 6.1: Packing of G.726-encoded signals (right-aligned, parallel format).

right-justified 2-, 3-, 4-, or 5-bit wide G.726 ADPCM samples, depending on the rate used.

smpno Is the number of samples in *inp_buf*.

law Is a char indicating if the law for the input samples is A ('1') or μ ('0'). See note below.

rate Is a short indicating the number of bits per sample to used by the algorithm: 5, 4, 3, or 2.

reset Is the reset flag (see note below):

- 1: reset is to be applied in the variables;
- 0: processing is carried out without setting state variables to the reset state.

Please note that this should normally be done only in the first call to the routine in processing a sample stream.

state The state variable structure; all the variables here are for internal use of the G.726 algorithm, and should not be changed by the user. Fields of this structure are described above.

Note: Please note the difference between *reset* and *law*: *reset* must be either 1 (0x01) or 0 (0x00), not '1' (0x31) or '0' (0x30), while *law* is exactly the opposite.

Return value: None.

6.2.2 G726_decode

Syntax:

```
#include "g726.h"
void G726_decode (short *inp_buf, short *out_buf, long smpno, char
                  *law, short rate, short reset, G726_state *state)
```

Prototype: g726.h

Description:

Simulation of the ITU-T G.726 ADPCM decoder. Takes the ADPCM input array of shorts *inp_buf* (16 bit, right-justified, without sign extension) of length *smpno*, and saves the decoded samples (A or μ law) in the array of shorts *out_buf*, with the same number of samples and right-justified.

The state variables are saved in the structure *state*, and the reset can be established by making *reset* equal to 1. The law is A if *law*== '1', and mu law if *law*== '0'.

Variables:

<i>inp_buf</i>	Is the input samples' buffer; each short sample will contain right-justified 2-, 3-, 4-, or 5-bit wide G.726 ADPCM samples.
<i>out_buf</i>	Is the output samples' buffer; each short sample shall contain right-justified 8-bit wide valid A or μ law samples.
<i>smpno</i>	Is the number of samples in <i>inp_buf</i> .
<i>law</i>	Is a char indicating if the law for the input samples is A ('1') or μ ('0'). See note below.
<i>rate</i>	Is a short indicating the number of bits per sample to used by the algorithm: 5, 4, 3, or 2.
<i>reset</i>	Is the reset flag (see note below): <ul style="list-style-type: none"> • 1: reset is to be applied in the variables; • 0: processing done without setting state variables to reset state. Please note that this should normally be done only in the first call to the routine in processing a sample stream.
<i>state</i>	The state variable structure; all the variables here are for internal use of the G.721 algorithm, and should not be changed by the user. Fields of this structure are described above.

Note: Please note the difference between *reset* and *law*: *reset* must be either 1 (0x01) or 0 (0x00), not '1' (0x31) or '0' (0x30), while *law* is exactly the opposite.

Return value: None.

6.3 Portability and compliance

Code testing has been done using the reset test sequences for 40, 32, 24, and 16 kbit/s provided in the G.726 test sequence diskettes (available from the ITU sales department). Other tests were also done with speech files for the 32 kbit/s mode, comparing with reference implementations, most noticeably the one from AT&T Bell Laboratories, which is the original implementation. Both test approaches generated 100% compatibility of this implementation with the G.726.⁴

The portability of the STL G.726 encoding function has been tested by feeding the routine with the reset test sequences of the G.726 test sequences diskettes (available from the ITU

⁴The problem with the A-law 40 kbit/s test vector *ri40fa.o* present in the STL96 has been solved in the STL2000.

Secretariat). As inputs, a binary version of the files `nrm.a`, `ovr.a`, `nrm.m`, `ovr.m` have been used for the 4 bit rates; the output of `G726_encoder` was then compared with a binary version of the files `nrdfa.i`, `rvdfa.i`, `nrdfm.i`, `rvdfm.i`, $rr = 16, 24, 32, 40$, accordingly for each input sequence and rate. The encoding routine passed the test when no differences in the bit streams were found.

The portability test of the decoding function was carried out by feeding this routine with the pertinent test sequences of the G.726 Test Sequences Diskettes. As inputs, a binary version of the files `nrdfa.i`, `rvdfa.i`, `nrdfm.i`, `rvdfm.i`, `nrdfm.i`, `rvdfm.i`, `nrdfm.i`, `rvdfm.i`, and `irr` (twice: one for A and another for μ law) have been used, rr being 16, 24, 32, and 40. The output of `G726_decoder` was then compared with a binary version of the files `nrdfa.o`, `rvdfa.o`, `nrdfx.o`, `rvdfx.o`, `nrdfm.o`, `rvdfm.o`, `nrdfc.o`, `rvdfc.o`, `nrdfa.o`, `nrdfm.o` (rr as above), respectively for each input sequences. All test vectors were properly processed.

These routines have been tested in VAX/VMS with VAX-C and GNU-C, in the PC with Borland C v3.0 (16-bit mode) and GNU-C (32-bit mode). In the Unix environment for Sun cc, acc, and gcc, and in HP for gcc.

6.4 Example code

6.4.1 Description of the demonstration programs

Two programs are provided as demonstration programs for the G.726 module, `g726demo.c` and `vbr-g726.c`.

Program `g726demo.c` accepts input files in either 16-bit, right-justified A- or μ -law format (as generated by `g711demo.c`) and encodes and/or decodes using one of the G.726 bit rates (16, 24, 32, or 40 kbit/s). Linear PCM files are not accepted by the program. Three operations are possible: logarithmic in, logarithmic out (*lolo*) logarithmic in, ADPCM out (*load*), or ADPCM in, logarithmic out (*adlo*).

Program `vbr-g726.c` can perform the same functions as `g726demo.c`, however it is capable of two additional features. It can perform in variable bit rate mode, which is switched at user-specified frame sizes (i.e. number of samples), and it can operate from 16-bit linear PCM input files. In the latter case, A-law is used to compand the linear signal prior to G.726 encoding, since G.726 Annex A [26] is not yet implemented in the STL.

6.4.2 Simple example

The following C code gives an example of G.726 coding and decoding using as input speech previously encoded by either the A- or μ -law functions available in the STL. The output samples will be encoded using the same law of the input signal.

```
#include <stdio.h>
#include "ugstdemo.h"
#include "g726.h"
```

```

#define BLK_LEN 256

void main(argc, argv)
    int      argc;
    char     *argv[];
{
    G726_state      encoder_state, decoder_state;
    char            law[4];
    short           bitrate, reset;
    char            FileIn[180], FileOut[180];
    short           tmp_buf[BLK_LEN], inp_buf[BLK_LEN], out_buf[BLK_LEN];
    FILE            *Fi, *Fo;

    /* Get parameters for processing */
    GET_PAR_S(1, "_Law: ..... ", law);
    GET_PAR_I(2, "_Bit-rate: ..... ", bitrate);
    GET_PAR_S(2, "_Input File: ..... ", FileIn);
    GET_PAR_S(3, "_Output File: ..... ", FileOut);

    /* Opening input and output LOG-PCM files */
    Fi = fopen(FileIn, RB);
    Fo = fopen(FileOut, WB);

    /* File processing */
    reset = 1;                      /* set reset flag as YES */
    while (fread(inp_buf, BLK_LEN, sizeof(short), Fi) == BLK_LEN)
    {
        /* Process input log PCM samples in blocks of length BLK_LEN */
        G726_encode(inp_buf, tmp_buf, BLK_LEN, law, bitrate, reset, &encoder_state);

        /* Process ADPCM samples in blocks of length BLK_LEN */
        G726_decode(tmp_buf, out_buf, BLK_LEN, law, bitrate, reset, &decoder_state);

        /* Write PCM output word */
        fwrite(out_buf, BLK_LEN, sizeof(short), Fo);

        if (reset)
            reset = 0;              /* set reset flag as NOMORE */
    }

    /* Close input and output files */
    fclose(Fi);
    fclose(Fo);
}

```

Chapter 7

G.727: The ITU-T embedded ADPCM algorithm at 40, 32, 24, and 16 kbit/s

7.1 Description of the Embedded ADPCM

The G.727 algorithm is specified in ITU-T Recommendation G.727 [28] with the block diagram shown in Figure 7.1, and will not be further described here. Additional information can be found in [25], where a thorough comparison is made between different ADPCM schemes, including G.726 and G.727. Details on the linear interface for the G.727 algorithm are found in G.727 Annex A [29].

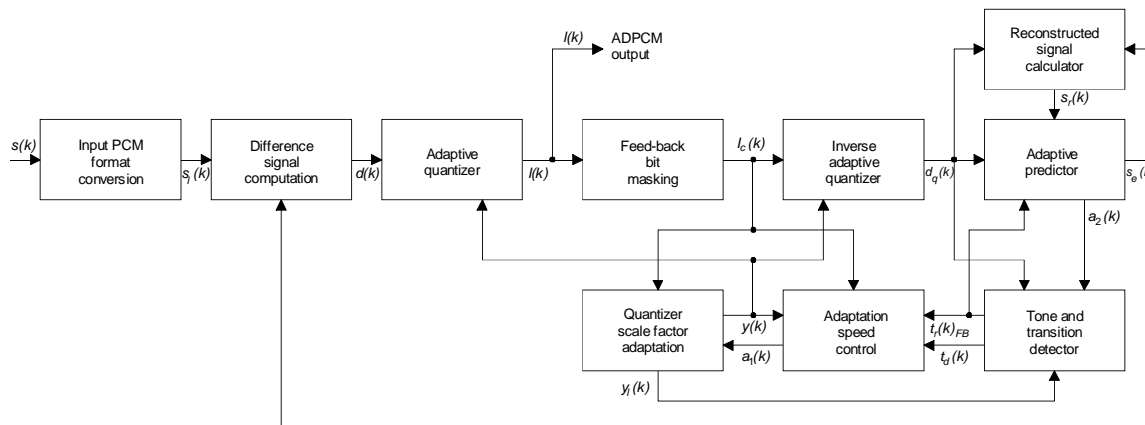
7.1.1 Extension for linear input and output signals

An extension of the G.727 algorithm was carried out in 1994 to include, as an option, linear input and output signals. The specification for such linear interface is given in its Annex A [29].

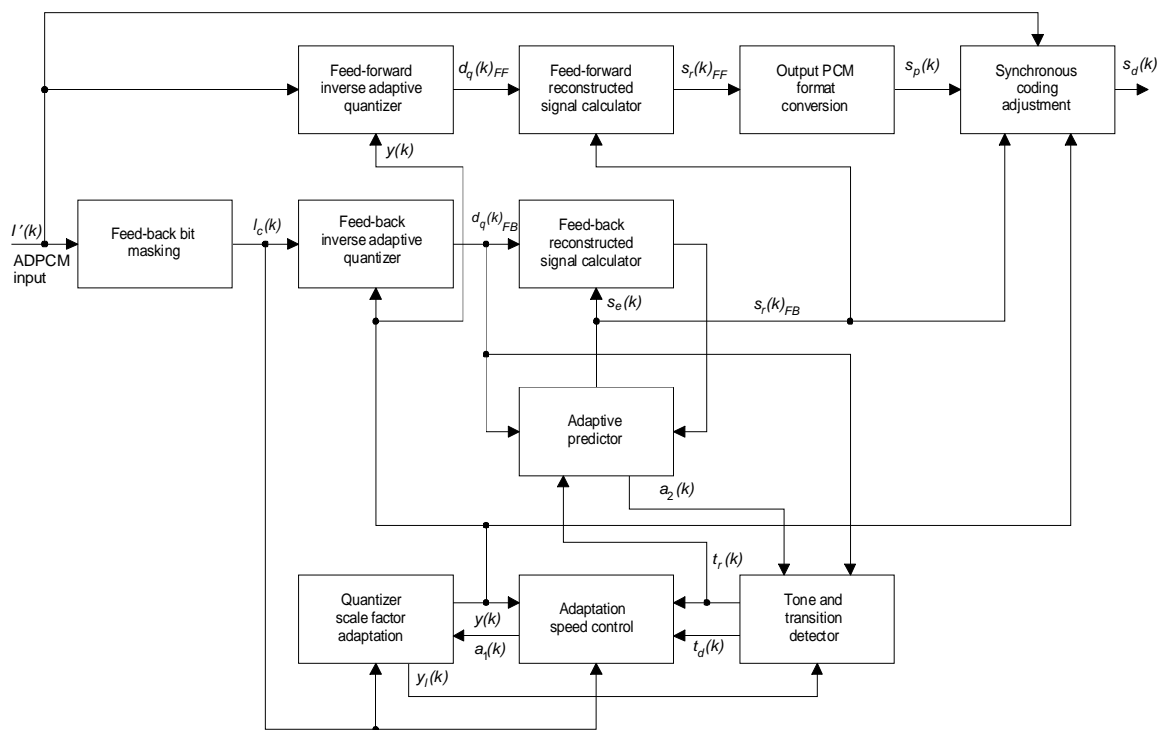
This extension bypasses the PCM format conversion block for linear input signals, and both the Output PCM Format Conversion and the Synchronous Coding Adjustment blocks, for linear output signals. These linear versions of the input and output signals are 14-bit, 2's complement samples.

The effect of removing the PCM encoding and decoding is to decrease the coding degradation by 0.6 to 1 qdu, depending on the network configuration considered (presence or absence of a G.712 filtering).

Currently, this extension has not been incorporated in the STL.



(a) Encoder



(b) Decoder

Figure 7.1: G.727 encoder and decoder block diagrams

7.2 ITU-T STL G.727 Implementation

The STL implementation of the G.727 algorithm can be found in module `g727.c`, with prototypes in `g727.h`.

The problem of storing the state variables was solved by defining a structure containing all the necessary variables, defining a new type called `G727_state`. As for other STL modules, the use of the state variable allows for parallel processing flows in the same executable program. The internal elements of the state variable `G727_state` should not be modified by the user, and are not described here.

The encoding function is `G727_encode`, and the decoding function is `G727_decode`. Additionally, initialization and reset of the state variable is performed by `g727_reset`. There are other internal routines which are not for access by the user, and hence are not described here. Their usage description is given below.

7.2.1 G727_reset

Syntax:

```
#include "g727.h"
void G727_reset (g727_state *st);
```

Prototype: `g727.h`

Description:

Reset ITU-T G.727 embedded ADPCM encoder or decoder state variable.

7.2.2 G727_encode

Syntax:

```
#include "g727.h"
void G727_encode (short *src, short *dst, short smpno, short law,
                 short cbits, short ebits, g727_state *state);
```

Prototype: `g727.h`

Description:

Simulation of the ITU-T G.727 embedded ADPCM encoder. Takes the A or μ law input array of shorts `src` (16 bit, right-justified, without sign extension) of length `smpno`, and saves the encoded samples in the array of shorts `dst`, with the same number of samples and right-justified. The ADPCM samples will have `cbits` core bits, and `ebits` enhancement bits.

The state variables are saved in the structure `state`, which should be initialized by `g727_reset()` before use. A-law is used if `law=='1'`, and μ -law if `law=='0'`.

Variables:

`src` Is the input samples' buffer; each `short` sample shall contain

		right-justified 8-bit wide valid A or μ law samples.
<i>dst</i>	Buffer with right justified short ADPCM-encoded samples with cbits core bits and ebits enhancement bits. Unused MSBs are set to zero.
<i>smpno</i>	Is a short indicating the number of samples to encode.
<i>law</i>	Is a char indicating if the law for the input samples is A ('1') or μ ('0').
<i>cbits</i>	Number of core ADPCM bits.
<i>ebits</i>	Number of enhancement ADPCM bits.
<i>state</i>	The state variable structure; all the variables here are for internal use of the G.727 algorithm, and should not be changed by the user.

Return value: None.

7.2.3 G727_decode

Syntax:

```
#include "g727.h"
void G727_decode (short *src, short *dst, short smpno, short law,
                  short cbits, short ebits, g727_state *state );
```

Prototype: g727.h

Description:

Simulation of the ITU-T G.727 embedded ADPCM decoder. Takes the ADPCM input array of shorts **src** (16 bit, right-justified, without sign extension) of length **smpno**, and saves the decoded samples (A or μ law) in the array of shorts **dst**, with the same number of samples and right-justified. The ADPCM samples must have **cbits** core bits, and **ebits** enhancement bits.

The state variables are saved in structure **st**, which should be initialized by **g727_reset()** before use. The law is A if *law*=='1', and μ law if *law*=='0'.

Variables:

<i>src</i>	Buffer with right justified short ADPCM-encoded samples with cbits core bits and ebits enhancement bits. Unused MSBs are zero.
<i>dst</i>	Is the input samples' buffer; each short sample shall contain right-justified 8-bit wide valid A or μ law samples.
<i>smpno</i>	Is a short indicating the number of samples to encode.
<i>law</i>	Is a char indicating if the law for the input samples is A ('1') or μ ('0').
<i>cbits</i>	Number of core ADPCM bits.
<i>ebits</i>	Number of enhancement ADPCM bits.
<i>state</i>	The state variable structure; all the variables here are for internal use of the G.727 algorithm, and should not be changed by the user.

Return value: None.

7.3 Portability and compliance

Code testing has been done using the reset test sequences for 5, 4, 3, and 2 bits with the valid combination of core and enhancement bits. The reset test sequences can be acquired from the ITU Sales Department, and are not distributed with the STL. The testing procedure is implemented in the makefiles, which use a binary version of the test vectors. The implementation passed the compliance test when no differences were found between tested and reference test vectors. All test vectors were verified to be properly processed.

These routines have been tested in in MS-DOS with Turbo C++ v1.0 (16-bit mode) and GNU-C (go32 32-bit mode), and in Windows/32 with MS Visual C and CYGNUS/gcc. In the Unix environment, they have been tested for SunOs (cc, acc, and gcc), HP-UX (gcc), and Ultrix 4.0 (cc and gcc).

7.4 Example code

7.4.1 Description of the demonstration program

One program is provided as demonstration program for the G.727 module, `g727demo.c`.

Program `g727demo.c` accepts input files in either 16-bit, right-justified A- or μ -law format (as generated by `g711demo.c`) and encodes and/or decodes using the G.727 algorithm for the user-specified number of N_c core bits and N_e enhancement bits. The effective encoding bitrate will then be $16 \times (N_c + N_e)$ kbit/s. Linear PCM files are not accepted by the program, since G.727 Annex A [29] is not yet implemented in the STL. Three operations are possible: logarithmic in, logarithmic out (default) logarithmic in, ADPCM out (option `-enc`), or ADPCM in, logarithmic out (option `-dec`).

7.4.2 Simple example

The following C code gives an example of G.727 coding and decoding using as input speech previously encoded by either the A- or μ -law functions available in the STL. The output samples are encoded using the same law of the input signal.

```
#include <stdio.h>
#include "ugstdemo.h"
#include "g727.h"

#define BLK_LEN 256

void main(argc, argv)
    int      argc;
    char     *argv[];
{
    G727_state      encoder_state, decoder_state;
```

```

char          law;
short         core, enh;
char          FileIn[180], FileOut[180];
short         tmp_buf[BLK_LEN], inp_buf[BLK_LEN], out_buf[BLK_LEN];
FILE          *Fi, *Fo;

/* Get parameters for processing */
GET_PAR_C(1, "_Law: ..... ", law);
GET_PAR_I(2, "_Core bits: ..... ", core);
GET_PAR_I(2, "_Enhancement bits: ..... ", enh);
GET_PAR_S(2, "_Log-PCM Input File: ..... ", FileIn);
GET_PAR_S(3, "_Log-PCM Output File: ..... ", FileOut);

/* Opening input and output LOG-PCM files */
Fi = fopen(FileIn, RB);
Fo = fopen(FileOut, WB);

/* Reset state variables */
g727_reset(&encoder_state);
g727_reset(&decoder_state);

/* File processing */
while (fread(inp_buf, BLK_LEN, sizeof(short), Fi) == BLK_LEN)
{
    /* Process input log PCM samples in blocks of length BLK_LEN */
    G727_encode(inp_buf, tmp_buf, BLK_LEN, law, core, enh, &encoder_state);

    /* Process ADPCM samples in blocks of length BLK_LEN */
    G727_decode(tmp_buf, out_buf, BLK_LEN, law, core, enh, &decoder_state);

    /* Write PCM output word */
    fwrite(out_buf, BLK_LEN, sizeof(short), Fo);
}

/* Close input and output files */
fclose(Fi);
fclose(Fo);
}

```


Chapter 8

G.722: The ITU-T 64, 56, and 48 kbit/s wideband speech coding algorithm

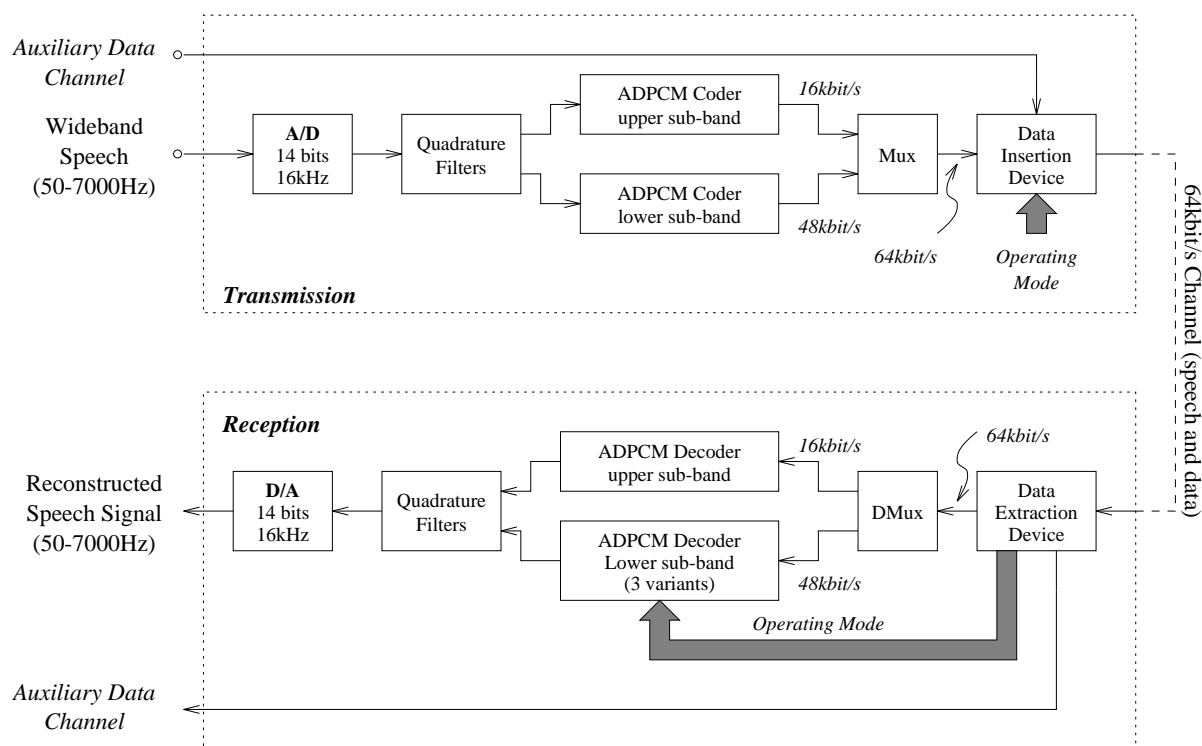
With the emergence of ISDN networks offering digital connectivity at 64 kbit/s between subscribers, the possibility was given to improve the standard telephone quality by increasing the transmitted bandwidth. A bandwidth of 50-7000 Hz corresponding to a sampling of 16 kHz was chosen because it provides a substantial improvement of the quality for applications where the speech is to be heard through high quality loudspeakers e.g. for audio or video conference services, commentary broadcasting, and high quality handsfree phones.

An expert group was created in November 1983 whose mandate was to define a single worldwide standard for 7 kHz speech coding within 64 kbit/s. After many contribution received from several organisations, it has been decided to choose a coder which combined subband filtering and adaptive differential pulse-code modulation algorithms (SB-ADPCM). The final recommendation was produced in March 1986 and approved in July 1986 by the then CCITT SG XVIII as Recommendation G.722 [30].

The full description on the implementation of the G.722 algorithm is found in [30], and network aspects related to its operation are found in [31]. Figure 8.1 summarizes some systemic aspects for the deployment of the G.722 algorithm. Overview and notes on the development of the G.722 algorithm can be found in several in [32, 33, 34, 35, 36, 37]. The following description of the G.722 algorithm is based on the text in [38].

8.1 Description of the 64, 56, and 48 kbit/s G.722 algorithm

In order to improve the transmitted speech quality, the input signal has to be converted after antialiasing filtering by an analog-to-digital (A/D) converter operating at a 16 kHz sampling rate and with a resolution of at least 14 uniform PCM bits. Similarly, at the receive side, a digital-to-analog (D/A) converter operating at a 16 kHz sampling

**Notes:**

* Operating modes:

- Mode 1:** 64kbit/s for speech and 0kbit/s for auxiliary data
- Mode 1-bis:** 56kbit/s for speech and 0kbit/s for auxiliary data
- Mode 2:** 56kbit/s for speech and 8kbit/s for auxiliary data
- Mode 3:** 48kbit/s for speech and 16kbit/s for auxiliary data
- Mode 3-bis:** 48kbit/s for speech, 6.4kbit/s for auxiliary data and 1.6kbit/s for service channel framing and mode control

* Operating modes 1-bis and 3-bis are applicable only to US national 56 kbit/s networks

* The signal in the 64kbit/s channel comprises 64, 56 or 48 kbit/s for speech and 0, 8 or 16 kbit/s for data, depending on the operating mode.

Figure 8.1: G.722 encoder and decoder block diagrams

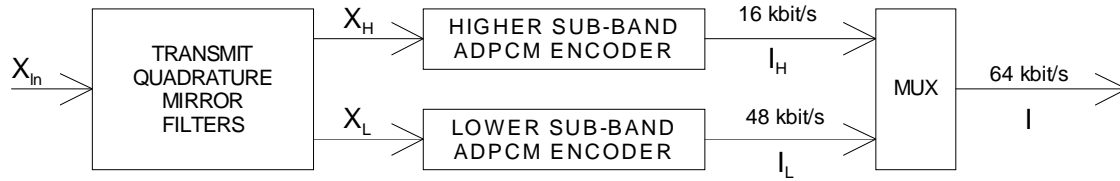


Figure 8.2: Block diagram of the SB-ADPCM encoder

rate and with a resolution of at least 14 uniform PCM bits should be used, followed by a reconstructing filter. The specifications of the transmission characteristics of the audio parts suited for the G.722 algorithm are described in the Recommendation. Some flexibility of the output bit rate was implemented to allow the opening of an auxiliary data channel within the 64 kbit/s channel.

8.1.1 Functional description of the SB-ADPCM encoder

Figure 8.2 shows a block diagram of the SB-ADPCM encoder which comprises the following main blocks.

Transmit quadrature mirror filters

The input signal x_{in} is first filtered by two quadrature mirror filters (QMF) which split the frequency band $[0, 8000 \text{ Hz}]$ into two equal subbands. The outputs x_l and x_h of the lower and higher subbands are downsampled at 8 kHz by the filtering procedure.

Lower subband ADPCM encoder

Figure 8.3 shows a block diagram of the lower subband ADPCM encoder. The encoder was designed to operate at 6, 5 or 4 bits per sample corresponding to 48, 40 or 32 kbit/s to transmit the lower band. The ADPCM algorithm is very similar to the embedded ADPCM algorithm of ITU-T Recommendation G.727 [28]. It is an embedded ADPCM with 4 core bits and 2 additional bits. The embedded property was introduced to prevent degradation in speech quality when the encoder and the decoder operate during short intervals in different modes.

Adaptive quantizer A 60-level non-uniform adaptive quantizer is used to quantize the difference el between the input signal x_l and the estimated signal sl . The output of the quantizer Il is the ADPCM codeword for the lower subband. The 4 forbidden output codewords were primarily introduced to prevent the generation of all zero codes at all modes, but have also later be used to recover the 8 kHz frame used by the coder.

Inverse adaptive quantizer In the feedback loop the two least significant bits of Il are deleted to produce a 4 bit signal Il_t which is used for the adaptation of the quantizer

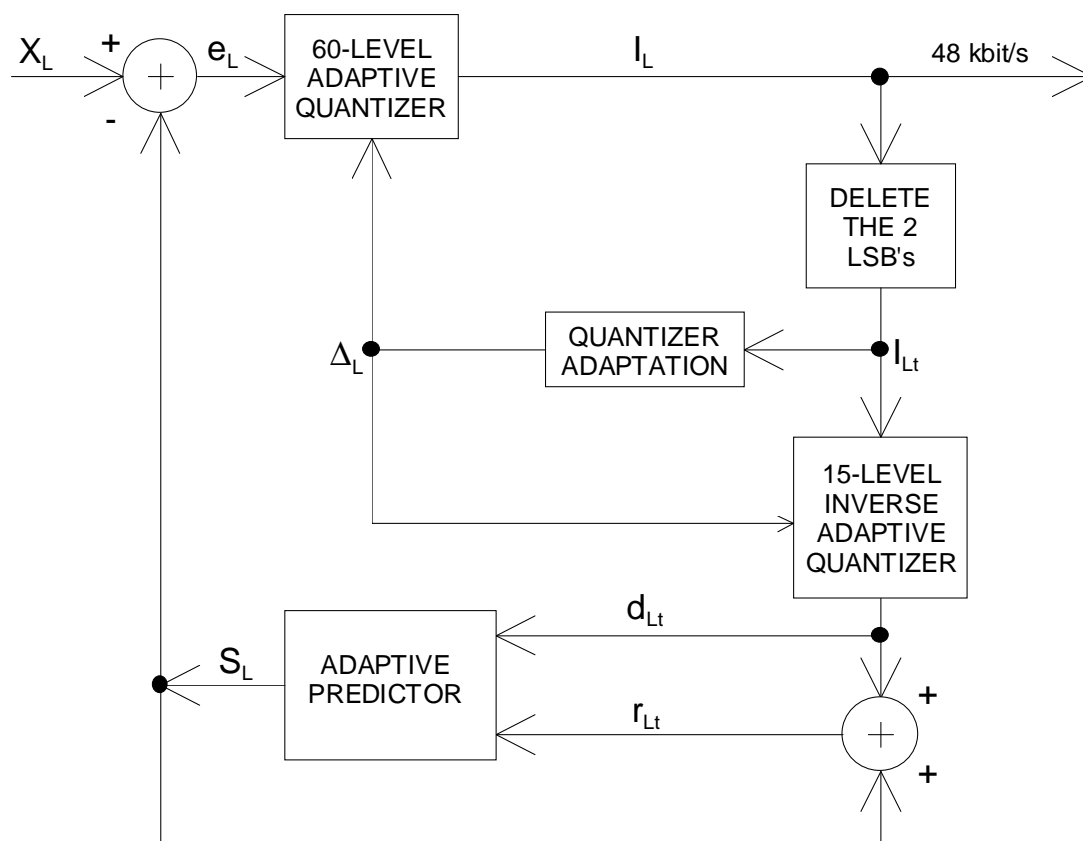


Figure 8.3: Block diagram of the lower subband ADPCM encoder

scale factor and applied to a 15-level inverse adaptive quantizer to produce the quantized difference signal dlt .

Quantizer adaptation In order to maintain a wide dynamic range and minimize complexity, the quantizer scale factor adaptation is performed in the base 2 logarithmic domain. The log-to-linear conversion is accomplished using a lookup table. There is no adaptation of the speed control parameter as in 32 kbit/s ADPCM [22] because the encoder is not designed to transmit voiceband data.

Adaptive predictor and reconstructed signal computation The adaptive predictor structure is similar to the one used for G.727 ADPCM standard: 2 poles and 6 zeroes. The two sets of coefficients (one for the poles and the other for the zeroes section) are updated using a simplified gradient algorithm. Stability constraints are applied to the poles in order to prevent possible unstable conditions. However, no predictor reset is applied for some specific inputs conditions as it is done in G.726 algorithm. The reconstructed signal rlt is computed by adding the quantized difference signal dlt to the signal estimate sl produced by the adaptive predictor. The use of a 4-bit operation instead of a 6-bit operation in the feedback loops of the lower band ADPCM encoder and decoder allows for the insertion of data in the two least significant bits without causing mistracking in the decoder.

Higher subband ADPCM encoder

Figure 8.4 shows a block diagram of the higher subband ADPCM encoder. This encoder is designed to operate at 2 bits per sample, corresponding to a fixed bit rate of 16 kbit/s. The encoder algorithm is very similar to the lower band one but with the following main differences. The quantizer is a 4-level non-linear adaptive quantizer. The higher subband ADPCM encoder is not embedded, hence the inverse quantizer uses the 2 bits in the feedback loop.

Multiplexer

The resulting codewords from the higher and lower subbands Ih and Il are combined to get the output codeword I with an octet format for transmission every 8 kHz frame producing a 64 kbit/s rate at the encoder output. Notice that the 8 kHz clock may be provided by the network as it is always done for 64 kbit/s A-law or μ -law log-PCM (G.711) systems.

8.1.2 Functional description of the SB-ADPCM decoder

Figure 8.5 shows a block diagram of the SB-ADPCM decoder.

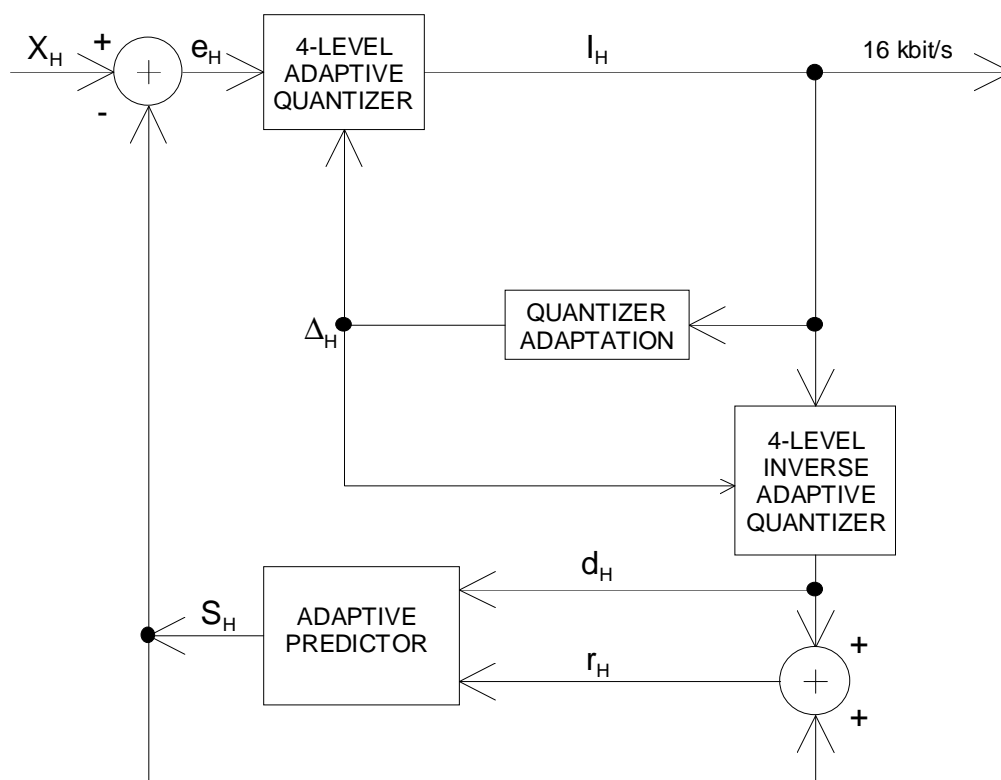


Figure 8.4: Block diagram of the higher subband ADPCM encoder

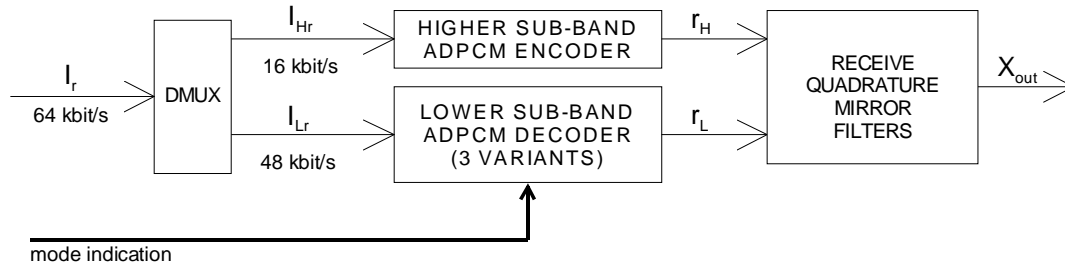


Figure 8.5: Block diagram of the SB-ADPCM decoder

Demultiplexer

The demultiplexer decomposes the received 64 kbit/s octet formatted signal I_r into two signals I_{lr} and I_{hr} which form the codeword inputs for the lower and higher subband ADPCM decoders.

Lower subband ADPCM decoder

Figure 8.6 shows a block diagram of the lower subband decoder. This decoder can operate in three different modes depending on the received mode indication and corresponding to 64, 56 or 48 kbit/s. The block which produces the estimate signal is identical to the feedback portion of the lower subband ADPCM encoder. The reconstructed signal rl is produced by adding the signal estimate to the relevant quantized difference signals $dl6$, $dl5$ or $dl4$, which are selected according to the received indication of the mode of operation.

Higher subband ADPCM decoder

This decoder (see Figure 8.7) is identical to the feedback portion of the higher subband ADPCM encoder which is described in the section 8.1.1, the output being the reconstructed signal rh .

Receive QMF

The receive QMF are two reconstruction filters which interpolate the outputs of the lower and higher subband ADPCM decoders from 8 to 16 kHz (rh and rl) and generate the global reconstructed output $xout$ sampled at 16 kHz. Signal $xout$ is converted to analog by the digital to analog converter of the receiving side.

8.2 ITU-T STL G.722 Implementation

This implementation of the G.722 algorithm is composed of several source files. The interface routines are in file `g722.c`, with prototypes in `g722.h`. The original code of

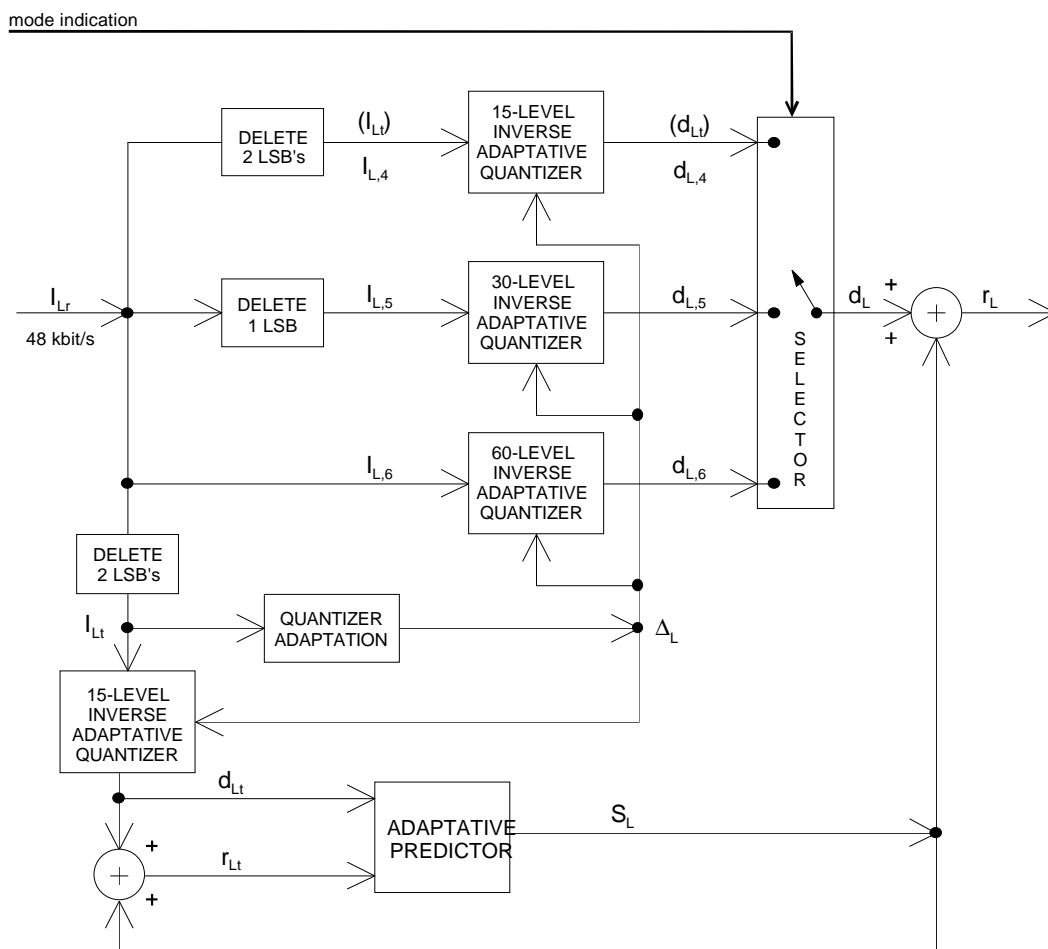


Figure 8.6: Block diagram of the lower subband ADPCM decoder

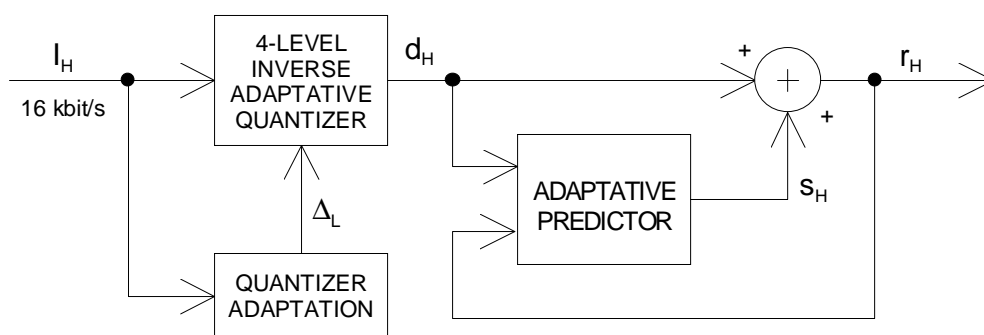


Figure 8.7: Block diagram of the higher subband ADPCM decoder

the STL G.722 was provided by CNET/France and its user interface was modified to be consistent with the other software modules of the STL.

The problem of storing the state variables was solved by defining a structure called `g722_state` which containing all the necessary state variables. By means of this approach, several streams may be processed in parallel¹, provided that one structure is assigned (and that one call to the encoding/decoding routines is done) for each data stream (this can be advantageous for machines with support for parallel processing). The G.722 state structure has the following fields (which are all `shorts`):

<i>ah, al</i>	Second-order pole section coefficient buffer for higher and lower band, respectively
<i>bh, bl</i>	Seventh-order zero section coefficient buffer for higher and lower band, respectively
<i>deth, detl</i>	Delayed quantizer scale factor for higher and lower band, respectively
<i>dh</i>	Quantizer difference signal memory
<i>dlt</i>	Quantizer difference signal for the adaptive predictor
<i>init_qmf_rx</i>	Flag indicating the need to initialize the QMF filters on the reception (decoder) side
<i>init_qmf_tx</i>	Flag indicating the need to initialize the QMF filters on the transmission (encoder) side
<i>nbh, nbl</i>	Delayed logarithmic quantizer factor for higher and lower band, respectively
<i>ph, plt</i>	Partially reconstructed signal memory for higher and lower band, respectively
<i>qmf_rx_delayx</i>	Memory of past 24 received (decoded) samples
<i>qmf_tx_delayx</i>	Memory of past 24 transmitted (encoded) samples
<i>rh[3]</i>	Quantized reconstructed signal
<i>rlt[3]</i>	Reconstructed signal memory for the adaptive predictor
<i>sh, sl</i>	Predictor output value for higher and lower band, respectively
<i>sph, spl</i>	Pole section output signal for higher and lower band, respectively
<i>szh, szl</i>	Zero section output signal for higher and lower band, respectively

The bitstream generated by the STL G.722 encoder has 8 valid bits for each encoded sample, saved in right-justified `shorts`. The lower 6 bits are the lower-subband encoded bits, and the upper two bits of the 8 valid bits are the upper-subband encoded bits. When the decoder is not in operation mode 1, the decoder will discard 1 or 2 of the lower bits of the lower-subband. It should be noted that, when bit errors are inserted in this bitstream and the operation mode is not mode 1, the actual bit error rate seen by the decoder may not be the one actually desired. One may consider that, in simulating a system where auxiliary data channels are used, such as modes 2 and 3, this is actually the desired behaviour, because errors hitting the auxiliary data will not affect the decoded speech quality. However, if simulation of modes 1-bis or 3-bis is intended, then the some

¹This feature was not possible with the original code provided by CNET and was added in the modifications of the user interface.

of the errors hitting the lower 1 (mode 1-bis) or 2 bits (mode 3-bis) will not be seen by the decoder, and the overall bit error rate will actually be smaller than the desired one. There are two possible approaches to circumvent this problem:

- the use of an external program to shift the bitstream samples one or two bits (respectively for modes 1-bis or 3-bis) to the right before the bitstream serialization process for use with the STL EID module, and an external program to left-shift the bitstream samples by one or two bits after error insertion and before using the STL G.722 decoder. This solution is valid for both random and burst bit errors.
- to increase proportionally the bit error rate by 1/8 (mode 1-bis) or 1/4 (mode 3-bis), to statistically compensate for errors hitting unused bits. This solution is valid only for random bit errors.

From the users' perspective, the encoding function is `g722_encode`, and the decoding function is `g722_decode`. Before using these functions, state variables for the encoder and the decoder must be initialized respectively by `g722_reset_encoder` and `g722_reset_decoder`. It should be noted that encoder and decoder need individual state variables to work properly.

In the following part a summary of calls to the three entry functions is found.

8.2.1 `g722_encode`

Syntax:

```
#include "g722.h"
void g722_encode (short *inp_buf, short *g722_frame, long smpno,
                 g722_state *g722_encode);
```

Prototype: `g722.h`

Description:

Simulation of the ITU-T G.722 64 kbit/s encoder. Takes the linear (16-bit, left-justified) input array of shorts `inp_buf` (16 bit, right-justified, without sign extension) with `smpno` samples, and saves the encoded bit-stream in the array of shorts `g722_frame`.

The state variables are saved in the structure pointed by `g722_encode`, and the reset can be established by making a call to `g722_reset_encoder`.

Variables:

<code>inp_buf</code>	Is the input samples' buffer with <code>smpno</code> left-justified 16-bit linear short speech samples.
<code>g722_frame</code>	Is the encoded samples' buffer; each short sample will contain the encoded parameters as right-justified 8-bit samples.
<code>smpno</code>	Is a long with the number of samples to be encoded from the input buffer <code>inp_buf</code> .
<code>g722_encode</code>	A pointer to the state variable structure; all the variables here are for internal use of the G.722 algorithm, and should not be changed by the user. Fields of this structure are described above.

Return value:

Returns the number of speech samples encoded.

8.2.2 g722_decode**Syntax:**

```
#include "g722.h"
void g722_decode (short *g722_frame, short *out_buf, int mode, long
                  smpno, g722_state *g722_decoder, );
```

Prototype: g722.h

Description:

Simulation of the ITU-T 64 kbit/s G.722 decoder. Reconstructs a linear (16-bit, left-justified) array of shorts *inp_buf* (16 bit, right-justified, without sign extension) with *smpno* samples from the encoded bit-stream in the array of shorts *g722_frame*.

The state variables are saved in the structure pointed by *g722_decoder*, and the reset can be established by making a call to *g722_reset_decoder*.

Variables:

<i>g722_frame</i>	Is the encoded samples' buffer; each short sample will contain the encoded parameters as right-justified 8-bit samples.
<i>out_buf</i>	Is the output samples' buffer with <i>smpno</i> left-justified 16-bit linear short speech samples.
<i>mode</i>	Is an int which indicates the operation mode for the G.722 decoder. If equal to 1, the decoder will operate at 64 kbit/s. If equal to 2, the decoder will operate at 56 kbit/s, discarding the least significant bit of the lower-band ADPCM. If equal to 3, the decoder will discard the two least significant bits of the lower band ADPCM, being equivalent to the 48 kbit/s operation of the G.722 algorithm. It should be noted that, for this implementation of the G.722 algorithm, mode 1-bis is identical to mode 2, and mode 3-bis is identical to mode 3.
<i>smpno</i>	Is a long with the number of samples in the input encoded sample buffer <i>g722_frame</i> to be decoded.
<i>g722_decoder</i>	A pointer to the state variable structure; all the variables here are for internal use of the G.722 algorithm, and should not be changed by the user. Fields of this structure are described above.

Return value:

Returns the number of speech samples encoded.

8.2.3 g722_reset_encoder**Syntax:**

```
#include "g722.h"
void g722_reset_encoder (g722_state *g722_encoder);
```

Prototype: g722.h

Description:

Initializes the state variables for the G.722 encoder or decoder. Coder and decoder require each a different state variable.

Variables:

g722_encoder. A pointer to the G.722 encoder state variable structure which is to be initialized.

Return value: None.

8.2.4 g722_reset_decoder

Syntax:

```
#include "g722.h"
void g722_reset_decoder (g722_state *g722_decoder);
```

Prototype: g722.h

Description:

Initializes the state variables for the G.722 decoder. Coder and decoder require each a different state variable.

Variables:

g722_decoder. A pointer to the G.722 decoder state variable structure which is to be initialized.

Return value: None.

8.3 Portability and compliance

The portability test for these routines has been done using the test sequences designed by the ITU-T for the G.722 algorithm (available from the ITU sales department). It should be noted that the G.722 test sequences are not designed to test the QMF filters, but only to exercise the upper and lower band encoder and decoder ADPCM algorithms. Therefore, testing of the codec with the test sequences was done with a special set of test programs that used the core G.722 upper- and lower-band ADPCM coding and decoding functions. All test sequences were correctly processed.

This module has been tested in VAX/VMS with VAX-C, in the PC with Turbo C++ v1.0 (16-bit mode) and GNU-C (32-bit mode), in the Unix environment in a Sun workstation with cc, and in HP with gcc.

8.4 Example code

8.4.1 Description of the demonstration programs

One demonstration program is provided for the G.722 module, `g722demo.c`. In addition, two programs are provided in the distribution when compliance testing of the encoder and decoder is necessary, `tstcg722.c` and `tstdg722.c`².

Program `g722demo.c` accepts 16-bit, linear PCM samples sampled at 16 kHz as encoder input. The decoder also produces files in the same format. The bitstream signals out of the encoder are always organized in 16-bit, right-justified words that use the lower 8 bits (i.e., 64 kbit/s). According to the user-specified mode, the decoder will decode the G.722-encoded bitstream using 64, 56, or 48 kbit/s (i.e. full 8 bits, discard 1 bit of the lower band, or discard 2 bits of the lower band).

8.4.2 Simple example

The following C code gives an example of G.722 coding and decoding using as input wideband speech which is encoded and decoded at either 64, 56, or 48 kbit/s, according to the user-specified parameter *mode*.

```
#include <stdio.h>
#include "ugstdemo.h"
#include "g722.h"
#define BLK_LEN 256

void main(argc, argv)
    int      argc;
    char      *argv[];
{
    g722_state      encoder_state, decoder_state;
    int             mode;
    char            FileIn[180], FileOut[180];
    short           smpno, tmp_buf[BLK_LEN], inp_buf[BLK_LEN], out_buf[BLK_LEN];
    FILE            *Fi, *Fo;

    /* Get parameters for processing */
    GET_PAR_S(1, "_Input File: ..... ", FileIn);
    GET_PAR_S(2, "_Output File: ..... ", FileOut);
    GET_PAR_I(3, "_Mode: ..... ", mode);

    /* Initialize state structures */
    g722_reset_encoder(&encoder_state);
    g722_reset_decoder(&decoder_state);

    /* Opening input and output 16-bit linear PCM speech files */
```

²The demonstration program `g722demo.c` cannot be used for compliance verification because the test vectors for G.722 do not foresee processing through the quadrature mirror filters.

```
Fi = fopen(FileIn, RB);
Fo = fopen(FileOut, WB);

/* File processing */
while (fread(inp_buf, BLK_LEN, sizeof(short), Fi) == BLK_LEN)
{
    /* Encode input samples in blocks of length BLK_LEN */
    smpno = g722_encode(inp_buf, tmp_buf, BLK_LEN, &encoder_state);

    /* Decode G.722-coded samples in blocks of length BLK_LEN */
    smpno = g722_decode(tmp_buf, out_buf, mode, smpno, &decoder_state);

    /* Write 16-bit linear PCM output decoded samples */
    fwrite(out_buf, smpno, sizeof(short), Fo);
}

/* Close input and output files */
fclose(Fi); fclose(Fo);
}
```

Chapter 9

RPE-LTP: The full-rate GSM codec

In 1988, the Groupe Special Mobile of the Conference Européenne des Postes et Telecommunications (CEPT) approved the first generation of a pan-European digital cellular radio system operating at a net rate of 13 kbit/s¹. Its speech coding algorithm, the RPE-LTP (Regular Pulse Excitation, Long Term Predictor) was a compromise solution of the two best coders at that stage. The full-rate GSM system started operation in the beginning of 1992 in some European countries and its expansion is expected in a mid-term. This coder, despite not being an ITU-T standard, is relevant for standardization studies when scenarios involving tandeming conditions between the PSTN and the European cellular system need to be studied.

The current version of the STL includes a RPE-LTP implementation based on a freely available implementation originally produced at the Technical Institute of the University of Berlin, for a Unix environment. This code has been adapted, corrected to work on several platforms, and tested with the recommended test vectors, all properly processed.

Details on the algorithm can be found in several references [39, 40, 41], besides the Recommendation itself [42].

9.1 Description of the 13 kbit/s RPE-LTP algorithm

The RPE-LTP is a frame based coder, encoding 20 ms frames of input data at a time. The encoder converts each 160 sample frame (8 kHz sampling rate, 13 bits uniform PCM format) into a bitstream frame of 260 bits. The decoder uses the 260 bitstream bits to generate a frame of 160 reconstructed speech samples.

9.1.1 RPE-LTP Encoder

A simplified block diagram of the RPE-LTP encoder [42] is shown in figure 9.1.

¹The GSM standard developed initially under the responsibility of the CEPT was later transferred to the European Standardisation Telecommunications Institute (ETSI), and the acronym GSM had its meaning changed to Global System for Mobile Communications. Currently, the GSM specifications are being maintained by the Third Generation Partnership Project, 3GPP (www.3gpp.org).

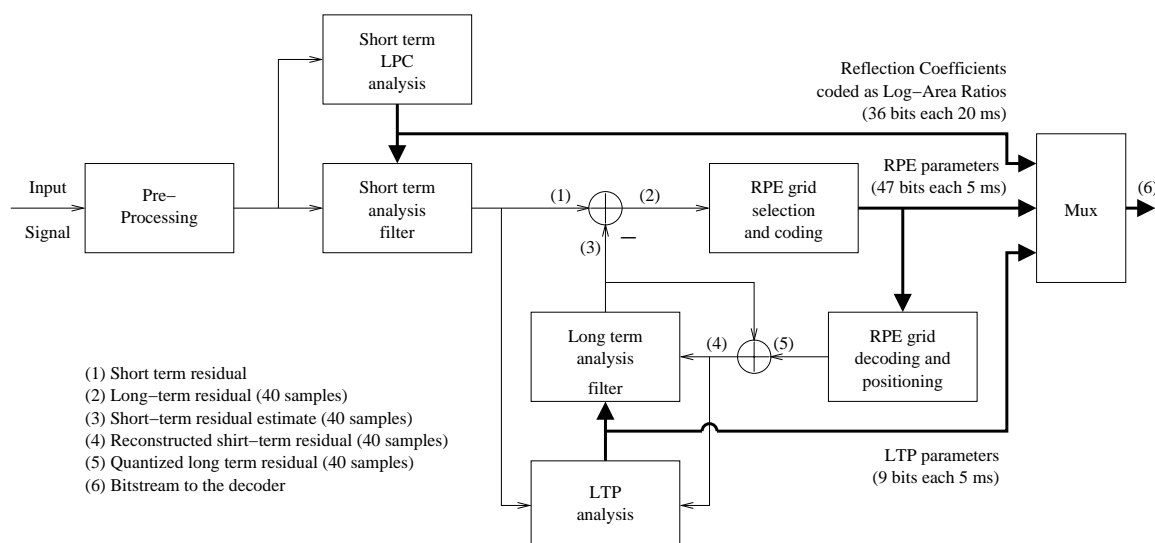


Figure 9.1: Simplified block diagram of the RPE-LTP encoder.

The input speech frame, consisting of 160 uniform 13 bits PCM signal samples, is first pre-processed to produce an offset-free signal, which is then subjected to a first-order pre-emphasis filter. The 160 samples obtained are then analyzed to determine the coefficients for the short-term analysis filter (LPC analysis). Using these coefficients for the filtering of the same 160 samples produce the 160 samples of the short-term residual signal. The filter parameters are represented as reflection coefficients which are transformed to log-area ratios (LARs) before transmission.

For the following operations, the speech frame is divided into 4 sub-blocks consisting each of 40 samples. Before the processing of each sub-block, the parameters of the long-term analysis filter, the LTP lag and the LTP gain, are estimated and updated in the LTP analysis block. Estimation and update is performed on the basis of the signal in the current sub-block and a stored sequence of the 120 previously reconstructed short-term residual samples.

A block of 40 long-term residual signal samples is obtained by subtracting 40 estimates of the short-term residual from the short-term residual signal itself. The resulting block is fed to the Regular Pulse Excitation (RPE) analysis which performs the basic compression function.

As a result of the RPE-analysis, the block of 40 input long-term residual samples is represented by one of 4 candidate sub-sequences of 13 pulses each. The subsequence selected is identified by the RPE grid position. The 13 RPE pulses are encoded using Adaptive Pulse Code Modulation (APCM) with estimation of the sub-block amplitude which is transmitted to the decoder as side information. The RPE parameters are also fed to a local RPE decoding and reconstruction module which produces a block of 40 samples of the quantized version of the long-term residual signal. By adding these 40 quantized samples of the long-term residual to the previously obtained block of short-term residual signal estimates, a reconstructed version of the current short-term residual signal is obtained. The block of reconstructed short-term residual signal samples is then fed to the long-term analysis filter which produces the new block of 40 short-term residual

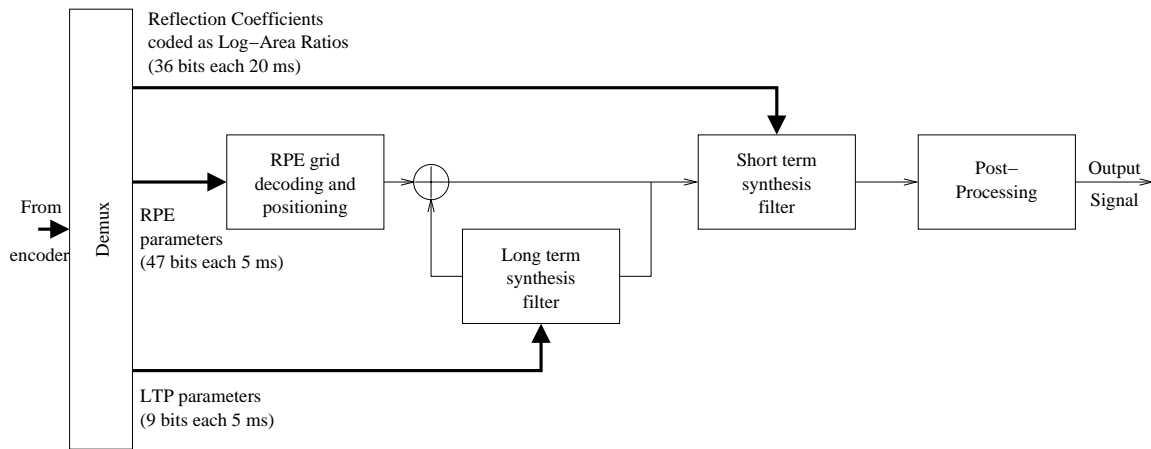


Figure 9.2: Simplified block diagram of the RPE-LTP decoder.

signal estimates to be used for the next sub-block thereby completing the feedback loop.

9.1.2 RPE-LTP Decoder

The simplified block diagram of the RPE-LTP decoder [42] is shown in figure 9.2.

The decoder includes the same structure as the feed-back loop of the encoder. In error-free transmission, the output of this stage will be the reconstructed short-term residual samples. These samples are then applied to the short-term synthesis filter followed by the de-emphasis filter resulting in the reconstructed speech signal samples.

9.2 Implementation

This implementation of the RPE-LTP algorithm is composed of several source files. The interface routines are in `rpeltp.c`, with prototypes in `rpeltp.h`.

Originally written to be a device driver in Unix (known as *toast*), its interface was adapted to the specifications of the ITU-T STL, and modified to operate correctly in a variety of platforms, like VAX, IBM PC compatibles, and Unix workstations (Sun and HP).

The problem of storing the state variables was solved by defining a structure containing all the necessary variables, defining a new type called `gsm`, which is a pointer to a structure. By means of this approach, several streams may be processed in parallel, provided that one structure is assigned (and that one call to the encoding/decoding routines is done) for each data stream (this can be advantageous for machines with support for parallel processing). The RPE-LTP state structure has the following fields (all except `L_z2` and `mp` are `short`, which are `long` and `int`, respectively):

<code>dp0</code>	Memory of 280 past samples
<code>z1</code>	DC-offset removal filter memory
<code>L_z2</code>	DC-offset removal filter parameter.
<code>mp</code>	Preemphasis

<i>u</i>	Eighth-order short term LPC analysis coefficients
<i>LARpp</i>	Log Area Ratio array
<i>j</i>	Index
<i>nrp</i>	Long-term synthesis parameter
<i>v</i>	Ninth order short-term synthesis vector
<i>msr</i>	Post-processing parameter
<i>verbose</i>	Flag used only if compiled with NDEBUG==0
<i>fast</i>	Enables fast but inaccurate computation. Does not properly process the test sequences with this mode turned on.

Table 9.1 presents the RPE-LTP encoder output parameters in order of occurrence, with parameters defined in [42]. It should be noted that the bitstream file generated by the STL implementation of the RPE-LTP algorithm uses an unpacked format, as other codecs in the STL. Therefore, each of the 76 parameters indicated in table 9.1 occupy an unsigned, right-adjusted 16-bit word. Unlikely to the G.711 and G.726 algorithms, however, the number of significant bits per bitstream parameter is not the same for all the parameters, as can be seen from the table. An important implication is that the STL bit error insertion routines cannot be applied directly to the bitstream generated by the STL RPE-LTP encoder. This limitation is not a function of the EID module itself, but of the serialization and parallelization (S/P) routines `serialize_*` and `parallize_*` implemented in the Utility module, which are able only to handle bitstreams that have the same number of valid bits per sample. Solution to this problem still needs to be implemented in the STL. It should be noted however that, since the full-rate GSM channel coding is not implemented in the STL, bit error insertion directly in the unprotected RPE-LTP bitstream will generally not be used. Should the user need bit error insertion in the unprotected RPE-LTP bitstream, there are two possible solutions:

- it will be necessary to pack the bits for each parameter in such a way that, as seen by the S/P routines, each sample in the packed bitstream will have a constant number of valid bits per bitstream sample. Since there are 260 ($4 \times 5 \times 13$) bits for each frame, possible combinations are packed bitstreams with 65 16-bit words, of which the lower 4 bits are meaningful, or with 20 16-bit words, of which the lower 13 bits are meaningful. The former is preferred, despite the longer files generated.
- the user may modify the demonstration program to generate or accept (depending on whether it is an encoding or decoding operation) a serial bitstream format, as understood by the EID module, instead of a parallel bitstream format.

From the users' perspective, the encoding function is `rpeltp_encode`, and the decoding function is `rpeltp_decode`. Before using these functions, the state variable for either the encoder or the decoder must be initialized by `rpeltp_init`. It should be noted that encoder and decoder need individual state variables to work properly. After all the processing is performed, the memory allocated for the state variables can be freed by calling `rpeltp_delete`. The following sub-sections describe these four entry functions for the STL RPE-LTP module.

Table 9.1: RPE-LTP bitstream format for each 20 ms speech frame.

Parameter	Parameter Number	Number of Bits
LAR1	1	6
LAR2	2	6
LAR3	3	5
LAR4	4	5
LAR5	5	4
LAR6	6	4
LAR7	7	3
LAR8	8	3
Sub-frame No. 1		
LTP lag	9	7
LTP gain	10	2
RPE grid position	11	2
Block amplitude	12	6
RPE-pulse no. 1	13	3
...
RPE-pulse no. 13	25	3
Sub-frame No. 2		
LTP lag	26	7
LTP gain	27	2
RPE grid position	28	2
Block amplitude	29	6
RPE-pulse no. 1	30	3
...
RPE-pulse no. 13	42	3
Sub-frame No. 3		
LTP lag	43	7
LTP gain	44	2
RPE grid position	45	2
Block amplitude	46	6
RPE-pulse no. 1	47	3
...
RPE-pulse no. 13	59	3
Sub-frame No. 4		
LTP lag	60	7
LTP gain	61	2
RPE grid position	62	2
Block amplitude	63	6
RPE-pulse no. 1	64	3
...
RPE-pulse no. 13	76	3

9.2.1 rpeotp_encode

Syntax:

```
#include "rpeotp.h"
void rpeotp_encode (gsm rpe_state, short *inp_buf, short *rpe_frame);
```

Prototype: rpeotp.h

Description:

Simulation of the GSM full-rate RPE-LTP encoder. The 16-bit, left-justified linear-PCM input array of **shortsamples** *inp_buf* are processed by the RPE-LTP encoder and the encoded bit-stream is returned in the right-justified array of **shortsamples** *rpe_frame*, with one sample for each encoded parameter. The input frame has 160 samples and the encoded frame has 76 samples.

The state variables are saved in the structure pointed by *rpe_state*, previously initialized by a call to **rpeotp_init()**. The reset can be established by making a call to **rpeotp_init()**.

Variables:

<i>rpe_state</i>	A pointer to the state variable structure. All the variables here are for internal use of the RPE-LTP algorithm and should not be changed by the user. Fields of this structure are described above.
<i>inp_buf</i>	Is the linear-PCM input sample buffer which must have 160 left-justified 16-bit linear-PCM shortsamples . Only the 13 MSb are used.
<i>rpe_frame</i>	Is the encoded sample buffer. Each shortsample will contain the encoded parameters as right-justified samples. The actual number of significant bits per sample will depend on each parameter.

Return value: None.

9.2.2 rpeotp_decode

Syntax:

```
#include "rpeotp.h"
void rpeotp_decode (gsm rpe_state, short *rpe_frame, short *out_buf);
```

Prototype: rpeotp.h

Description:

Simulation of the GSM full-rate RPE-LTP decoder. The encoded bit-stream in the input array of right-justified **shortsamples** *rpe_frame* is used to reconstruct a block of the speech signal using the RPE-LTP decoder. The reconstructed speech block is returned in the 16-bit, left-justified linear-PCM output array of **shortsamples** *inp_buf*. The input frame has 76 samples and the decoded frame has 160 samples.

The state variables are saved in the structure pointed by *rpe_state*, previously initialized

by a call to `rpeltp_init()`. The reset can be established by calling `rpeltp_init()`.

Variables:

<i>rpe_state</i>	A pointer to the state variable structure. All the variables here are for internal use of the RPE-LTP algorithm and should not be changed by the user. Fields of this structure are described above.
<i>rpe_frame</i>	Is the encoded sample buffer, which must have 76 right-justified shortsamples . The actual number of bits per sample will depend on each parameter.
<i>out_buf</i>	Is the output samples buffer, which will contain 160 left-justified, 13-bit linear-PCM shortsamples . The three LSBs are set to zero.

Return value: None.

9.2.3 `rpeltp_init`

Syntax:

```
#include "rpeltp.h"
gsm rpeltp_init (void);
```

Prototype: `rpeltp.h`

Description:

Initializes the state variables for the RPE-LTP encoder or decoder. Combined coder and decoder operation requires a different state variable for the encoding and the decoding part.

Variables: None.

Return value:

A pointer to an initialized state variable structure defined by the type `gsm`. Returns `NULL` in case of failure.

9.2.4 `rpeltp_delete`

Syntax:

```
#include "rpeltp.h"
void rpeltp_delete (gsm rpe_state);
```

Prototype: `rpeltp.h`

Description:

Releases memory allocated to a state variable previously initialized by `rpeltp_init()`.

Variables:

<i>rpe_state</i>	A pointer to a previously initialized RPE-LTP state variable structure.
------------------------	---

Return value:

None.

9.3 Portability and compliance

The portability test for these routines has been done using the test sequences designed by the GSM for the RPE-LTP (available from ETSI), which were also used to verify the compliance of the encoding and decoding function to the full-rate GSM voice codec Recommendation [42, Annex C].

This routine has been tested in VAX/VMS with VAX-C and gcc, in the PC with Borland C v3.0 (16-bit mode) and gcc (32-bit mode). In the Unix environment in a Sun workstation with cc, acc, and gcc, and in HP with gcc. In all tested cases, 100% of the test sequences passed when the following symbols were defined at compilation time: **SASR**, **USE_FLOAT_MUL** and **NDEBUG**. The symbol *FAST* must not be defined for performance compliant with the GSM 06.10 Recommendation, while **USE_FLOAT_MUL** **must** be defined at compilation time. The symbol **NeedFunctionPrototypes** must be undefined for pre-ANSI-C compilers (e.g. SunOS cc compiler).

9.4 Example code

9.4.1 Description of the demonstration program

One program is provided as demonstration program for the RPE-LTP module, `rpdemo.c`.

Program `rpdemo.c` accepts input files in either 16-bit linear PCM format, 16-bit, right-justified A-law format, or 16-bit, right-justified μ -law format for the encoding operation. The output of the decoder can also be in any of these formats, but it will have the same format as the encoding operation if encoding and decoding is performed in a single pass (default). If the encoding and decoding operations are performed in separate steps, the format of the output signal does not need to match the format of the original linear PCM signal. The encoder output and decoder input are signals in 16-bit, right-justified samples, as described before in Sections 9.2.1 and 9.2.2. Three operations are possible: encode and decode in a single pass (default), encode-only (option *-enc*), or decode-only (option *-dec*).

9.4.2 Simple example

The following C code gives an example of RPE-LTP coding and decoding using as input 13-bit, linear-PCM speech samples, which are encoded and decoded at 13 kbit/s.

```
#include <stdio.h>
#include "ugstdemo.h"
#include "rpeltp.h"
```

```

#define BLK_LEN 160

int main(argc, argv)
    int      argc;
    char     *argv[];
{
    gsm       encoder_state, decoder_state;

    char      FileIn[180], FileOut[180];
    short     bs_buf[BLK_LEN], inp_buf[BLK_LEN], out_buf[BLK_LEN];
    FILE      *Fi, *Fo;

    /* Get parameters for processing */
    GET_PAR_S(1, "_Input File: ..... ", FileIn);
    GET_PAR_S(2, "_Output File: ..... ", FileOut);

    /* Initialize state structures */
    encoder_state = rpeltp_init();
    decoder_state = rpeltp_init();

    /* Opening input and output LOG-PCM files */
    Fi = fopen(FileIn, RB);
    Fo = fopen(FileOut, WB);

    /* File processing */
    reset = 1;                      /* set reset flag as YES */
    while (fread(inp_buf, BLK_LEN, sizeof(short), Fi) == BLK_LEN)
    {
        /* Encode input linear PCM samples */
        rpeltp_encode(encoder_state, inp_buf, bs_buf, BLK_LEN);

        /* Decode samples */
        rpeltp_decode(decoder_state, bs_buf, out_buf);

        /* Write decoded samples */
        fwrite(out_buf, BLK_LEN, sizeof(short), Fo);

        if (reset)
            reset = 0;              /* set reset flag as NOMORE */
    }

    /* Free memory */
    rpe_delete(decoder_state);
    rpe_delete(encoder_state);

    /* Close input and output files */
    fclose(Fi);
    fclose(Fo);
    return 0;
}

```


Chapter 10

Duo-MNRU: The Dual-mode Modulated Noise Reference Unit

For evaluation of the quality of a system or equipment, it is important to express the quality measure in a unit suitable for comparison with other reference (or well-known) equipments and systems. A common way of representing these figures is by means of relative units, where the quality is expressed by means of a unique figure, in a unidimensional scale.

But it is insufficient to be unidimensional; the scale must be unequivocal, with a universal meaning. As an example, the ACR scale (*Absolute Category Rating*, [43, Annex B]), which is a scale used for listening opinion tests and has five points termed *Excellent*, *Good*, *Fair*, *Poor*, and *Bad*, is inadequate: besides it shows a continuum of quality points, the meaning of the adjectives are far from universal, varying from language to language, and from person to person. Exchange of information on the performance of these systems and equipments is easier and more consistent with more objective measures. The issue of how the MNRU is to be used as a reference system in subjective tests has been studied in ITU-T Study Group 12, which is described in ITU-T Recommendation P.830 [7] in its Sections 8.2.2 and 11.

The Modulated Noise Reference Unity (MNRU) was introduced as a means to controlled degradations that are representative of the non-linear distortion introduced by waveform coding techniques. Initially aiming at evaluating the quality of log-PCM waveform coding systems, it has been used in the process of generating several ITU-T standards, such as the ITU-T G.726 (32 kbit/s), G.722, G.728, and G.729.

The concept of such reference unit was published in [44]. The first system aimed at was the PCM coding with logarithmic compression (today world-wide available by means of the G.711 Recommendation), whose main characteristic is to have a considerably uniform signal-to-noise ratio (SNR) over a wide range of amplitudes. Moreover, the quantizing noise is correlated to the signal: if no signal is present, no quantization noise is produced¹, and large signals will produce more quantization noise than small ones. Therefore, the main characteristic of this reference unit should output speech corrupted by a speech-

¹This is obviously academic, because always there will be idle noise, among others, in the absence of an input signal.

correlated noise.

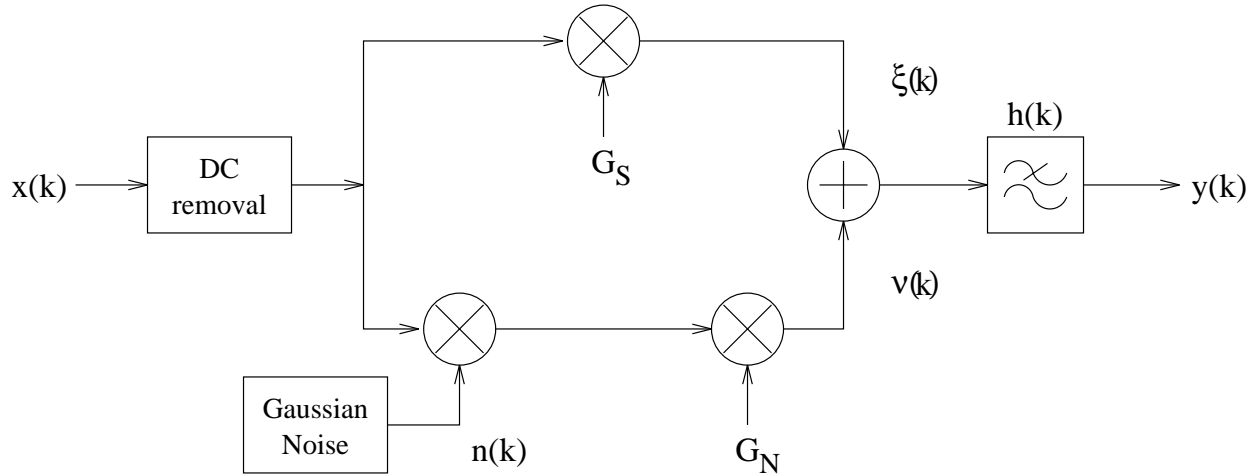


Figure 10.1: Block diagram of the “digital” MNRU. The bandwidth of the output filter $h(k)$ is 0–3400 Hz for the narrowband case, and 0–7000 Hz for the wideband case.

In [44], the speech-correlated noise generation was based on a double-balanced ring modulator, controlled by the input speech signal, which modulates a noise carrier generated by a noise generator having a relatively uniform energy distribution, there in the range of 0–20kHz. This correlated noise is then added to the input signal, with gains applied such that a controlled signal-to-noise ratio is obtained in the output, after the 300–3400Hz band-limiting filter.

With the 1996 revision of the MNRU description published in ITU-T Recommendation P.810², specific guidelines were given for “digital implementations”³, eliminating many of the ambiguities possible in earlier descriptions [45], as explained in the STL92 manual [46, Chapter 8]. Also, this implementation allows for transparent operation on narrowband or wideband speech, hence being known as Dual-mode MNRU, or “Duo-MNRU”, for short.

10.1 Description of the Algorithm

The de-facto reference implementation of the MNRU⁴ is the same of the original description, whose specification can be found in ITU-T Recommendation P.810 [47] (formerly ITU-T Recommendation P.81 [45]). This Recommendation describes two MNRU schemes, one called *Narrow-band MNRU*, and the other, *Wideband MNRU*. The latter is applicable to systems where wideband speech (70–7000Hz) is expected, whereas the former is for

²Formerly known as ITU-T Recommendation P.81.

³The revised P.81 define a “digital implementation” either as a digital hardware implementation or as a software implementation of the MNRU.

⁴Developed by the British Telecom and licenced to Malden Electronics.

telephone bandwidth (300–3400Hz). Both the narrowband and the wideband MNRU are implemented in this version of the ITU-T Software Tools Library.

The basic block diagram of the P.810 MNRU is found in figure 10.1. In summary, there are two paths, one called *signal path*, another called *noise path*. In the noise path, gaussian noise (uniform in a range at least the cutoff frequency of the low-pass filter in the output of the MNRU) is modulated by the incoming signal. The result is then added with the output from the signal path. The gains are set such that the gain (in dB) applied in the output of the noise path is the signal-to-correlated-noise ratio Q , in the output of the band-pass filter, as calculated in the section to follow.

In analytical terms, the signal corrupted by the modulated noise $y(k)$ is

$$y(k) = (G_s x(k) + G_n x(k)n(k)) * h(k)$$

where G_s is the gain of the signal path, G_n is the gain of the noise path, $x(k)$ is the input signal, and $n(k)$ is the gaussian noise signal; the symbol $*$ means convolution, and $h(k)$ is the band-pass filter.

If we suppose that the band-pass filter has $|H(f)| = 1$ in its pass band, and calling Q the signal-to-noise-ratio (SNR) at its output, we may write:

$$10^{Q/10} = \frac{\sigma_\xi^2}{\sigma_\nu^2} = \frac{E[\xi^2(k)]}{E[\nu^2(k)]} = \frac{G_s^2 E[x^2(k)]}{G_n^2 E[x^2(k)n^2(k)]}$$

But x and n are uncorrelated, and the noise is gaussian with mean 0 and variance 1 ($N(0,1)$):

$$10^{Q/10} = \left(\frac{G_s}{G_n}\right)^2 \frac{\sigma_x^2}{\sigma_x^2 \sigma_n^2} = \left(\frac{G_s}{G_n}\right)^2$$

or

$$\begin{aligned} Q &= \Gamma_s + \Gamma_n \\ \Gamma_s &= 20 \log_{10}(G_s) \\ \Gamma_n &= -20 \log_{10}(G_n) \end{aligned}$$

If we set $\Gamma_s = 0$ ($G_s = 1$), Q is exactly Γ_n (or, $G_n = 10^{-Q/20}$), i.e., the SNR is the gain (in dB) of the noise path and the previous expression may be written as:

$$y(k) = [x(n) + 10^{-Q/20} x(k)n(k)] * h(k)$$

or approximately

$$y(k) = x(k) + 10^{-Q/20} x(k)n(k)$$

in the passband region of $H(f)$.

When both G_s and G_n are non-zero, the MNRU is in an operational mode normally called *Modulated-noise mode*. This is the most common operation mode.

Alternatively, if one consider $G_s = 0$, the output of the algorithm is only the correlated noise, at a level Q dB below the input signal. This is *Noise-only mode*.

If, on the other hand, $G_n = 0$, the output of the algorithm is the input signal filtered by $h(k)$, with a gain G_s ; this is the *Signal-only mode*.

10.2 Implementation

This implementation of the MNRU algorithm can be found in the module `mnr.c`, with prototypes in `mnr.h`. A thorough characterization of this module is presented in [48]. The previous version of the ITU-T STL MNRU was applicable to narrowband signals and evolved from a Fortran implementation which had been used by several laboratories, especially by participants of ETSI's contest for the second generation of Digital Mobile Radio Systems, and was originally written by experts at CSELT/Italy (sometimes referred as *CSELT MNRU*), an implementation fully compliant with the narrowband MNRU specification available in the then-in-force P.81 [45].

With the revision of MNRU specification, several changes had to be made to the STL92 MNRU:

- The need for an upsampling by a factor of 5 before summation of the modulated noise to the input speech was eliminated because now for digital implementations, the bandwidth of the multiplicative noise shall have the bandwidth of the input signal. In the previous version, the noise bandwidth had to be 20 kHz.
- The output filter for digital implementations shall be a low-pass filter, instead of the bandpass filter of the previous version of the MNRU
- The need of an input speech DC-component removal filter was added to the specification.

These changes, especially the elimination of the 5:1 speech data rate conversion, allowed for the implementation of both the narrowband and the wideband MNRU within the same C function, when the output filter is adequately designed [48, pp.7–12].

The random number generator (RNG) algorithm was also modified to allow for real-time implementations, and the solution adopted was based on Aachen University's approach used by the Host Laboratory for the ITU-T G.729 Selection Tests.

The block diagram of the MNRU implemented in the STL96 is in figure 10.2.

The MNRU works internally on a sample-by-sample basis but for ease of interface with other speech coding functions, access to it is made on a sample block basis. It should be noted however that the filters have memory, as well as do the random number generator, hence state variables are needed. These state variables have been arranged as fields of a structure whose `type` name is `MNRU_state`. The fields of the structure are:

<i>seed</i>	RNG's seed
<i>signal_gain</i>	Gain of the signal path
<i>noise_gain</i>	Gain of the noise path
<i>vet</i>	Array for intermediate data
<i>last_xk</i>	$x(k-1)$ used as memory for the DC-removal filter
<i>last_yk</i>	$\xi(k-1)$ (see figure 10.2), used as memory for the DC-removal filter
<i>DLY[2][2]</i>	Memory of delayed samples for two second-order stages (first index) for first- and second-order delays (second index)

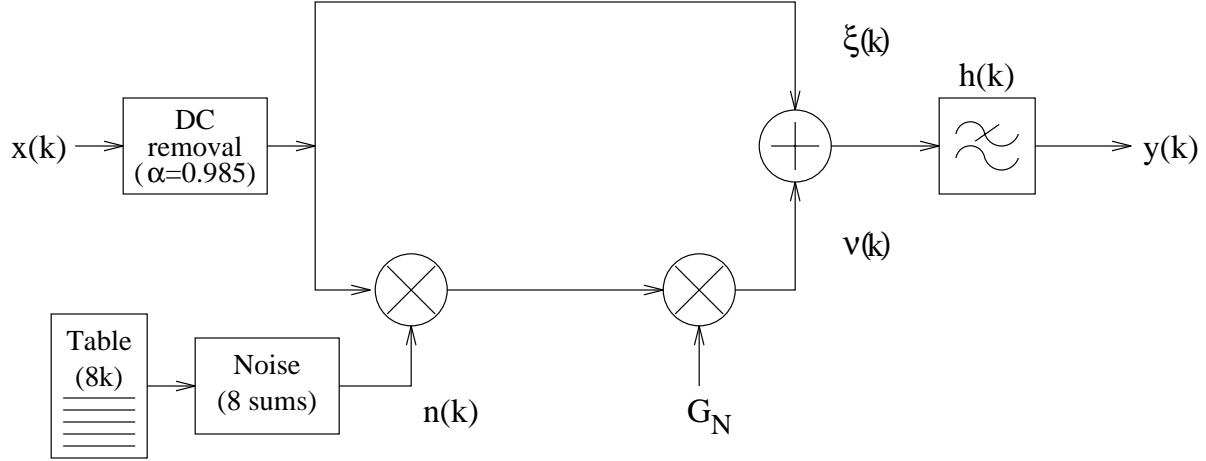


Figure 10.2: STL MNRU implementation.

$A[2][2]$	Numerator coefficients for the stage indicated by the first index and delay-order indicated by the second index
$B[2][2]$	Denominator coefficients for the stage indicated by the first index and delay-order indicated by the second index
rnd_state	State structure for MNRU's random number generator. Detailed description is found in the section on the random number generator.
rnd_mode	Operational mode of the random number generator
$clip$	Number of samples clipped in the noise-insertion process

The values of the fields shall not be altered by the user.

Filters in the MNRU module

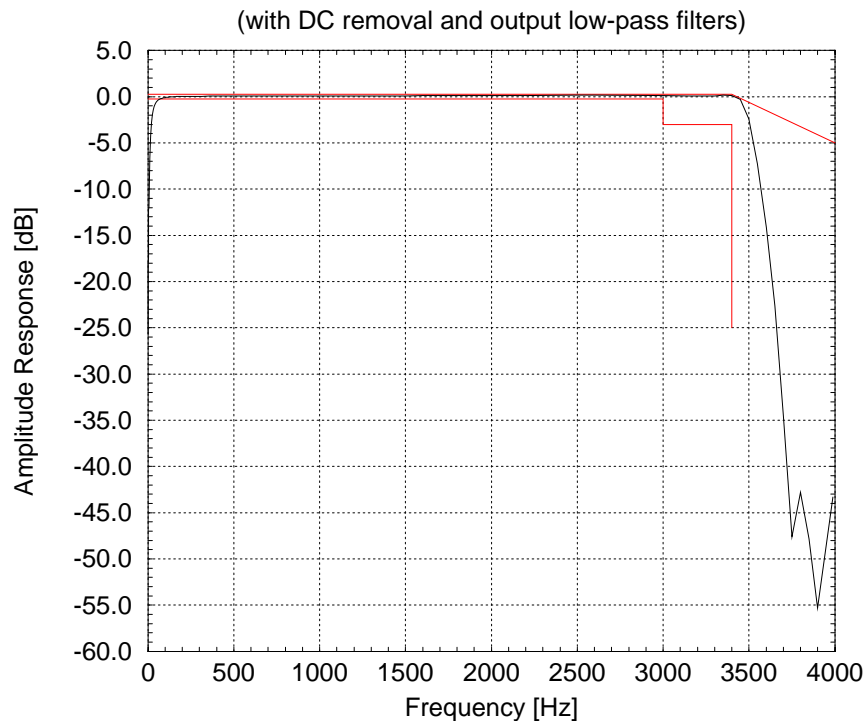
The composite frequency response of both MNRU filters is shown in figure 10.3. Figure 10.5 shows the contribution of the output low-pass filter for (a) the narrowband and (b) the wideband cases. Figure 10.4 shows the effect of the input DC-removal filter for (a) the narrowband and (b) the wideband operation modes of the MNRU. Details on the design of the output low-pass filters are given in [48]. The frequency responses have been obtained by exciting the MNRU module with digital sinewaves and computing the ratio of input and output signals, in dB.

The input DC-removal filter was implemented using a first-order IIR pole-zero filter defined by

$$H_i(z) = \frac{1 - z^{-1}}{1 - \alpha z^{-1}}$$

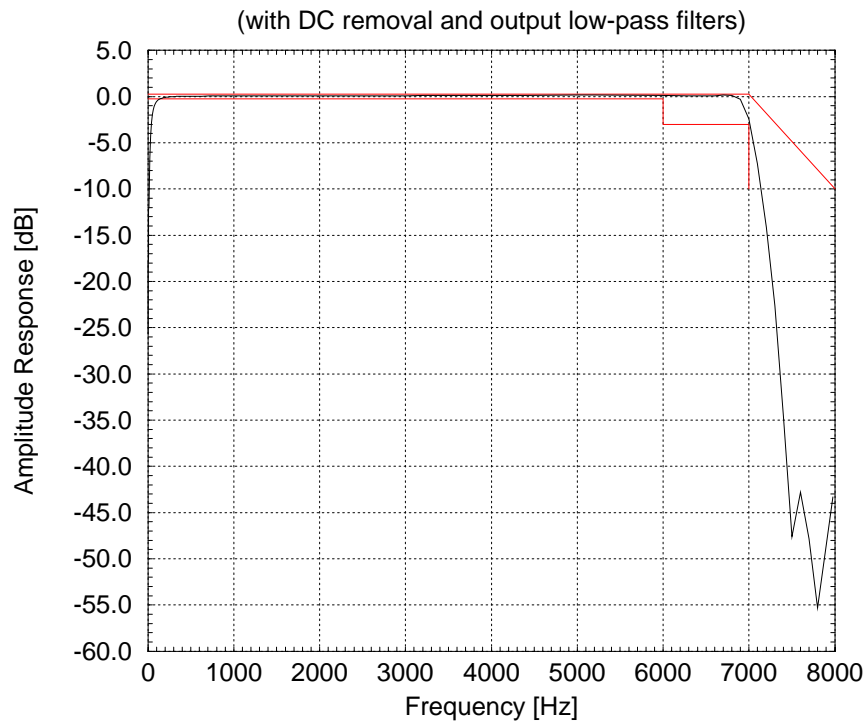
with $\alpha=0.985$. Its 3-dB point is at 16 Hz for the narrowband case and at 38 Hz for the

Duo-MNRU response for narrow-band signals



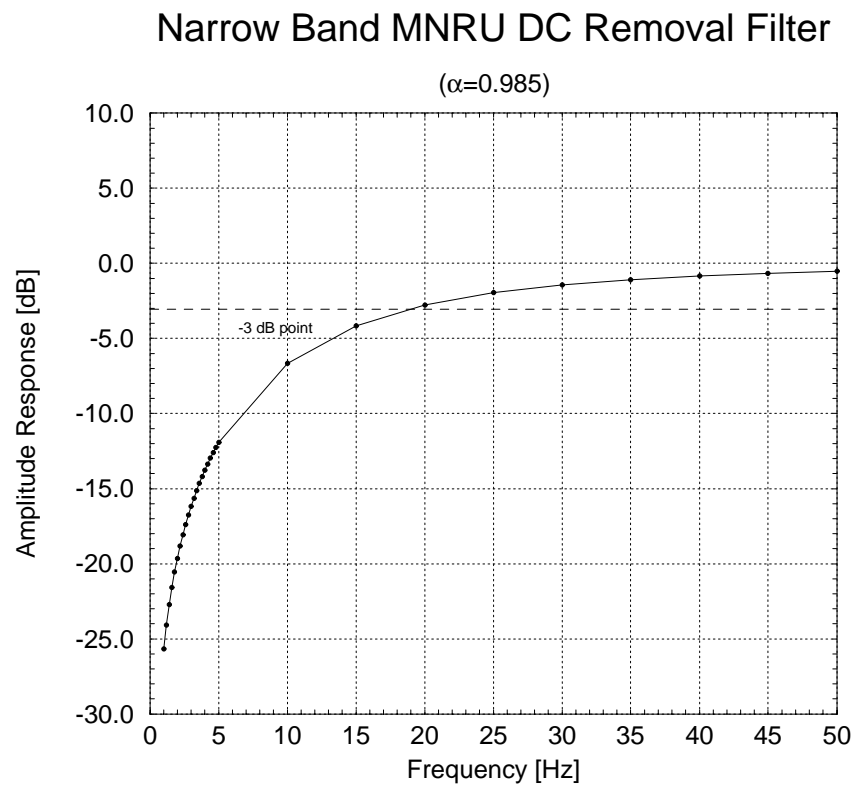
(a) Narrowband Duo-MNRU

Duo-MNRU response for wideband signals

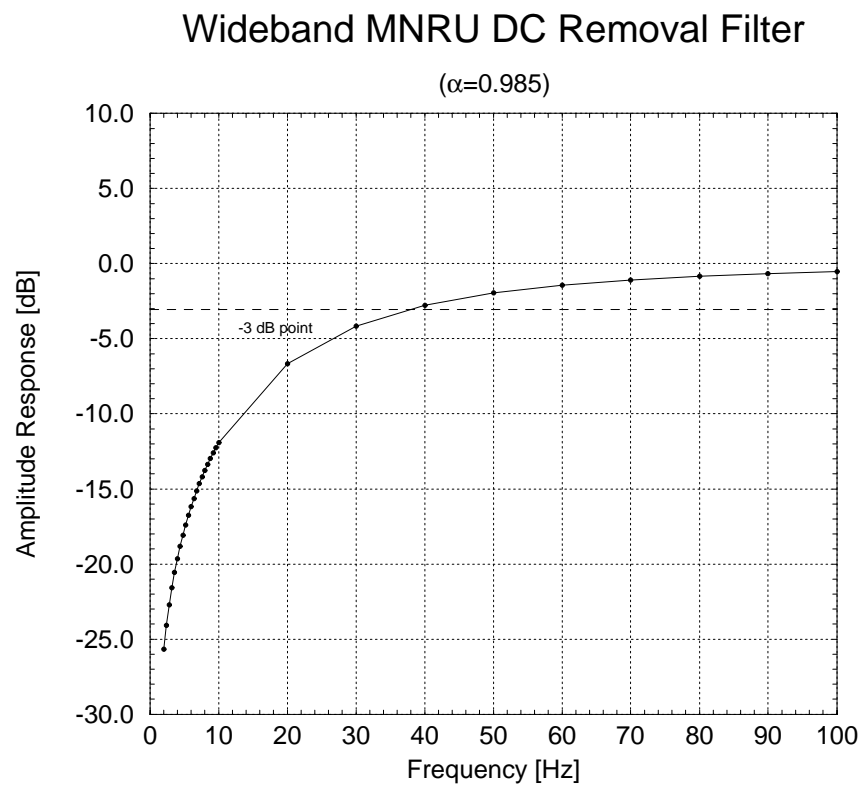


(b) Wideband Duo-MNRU

Figure 10.3: Total frequency response of the Duo-MNRU filters.



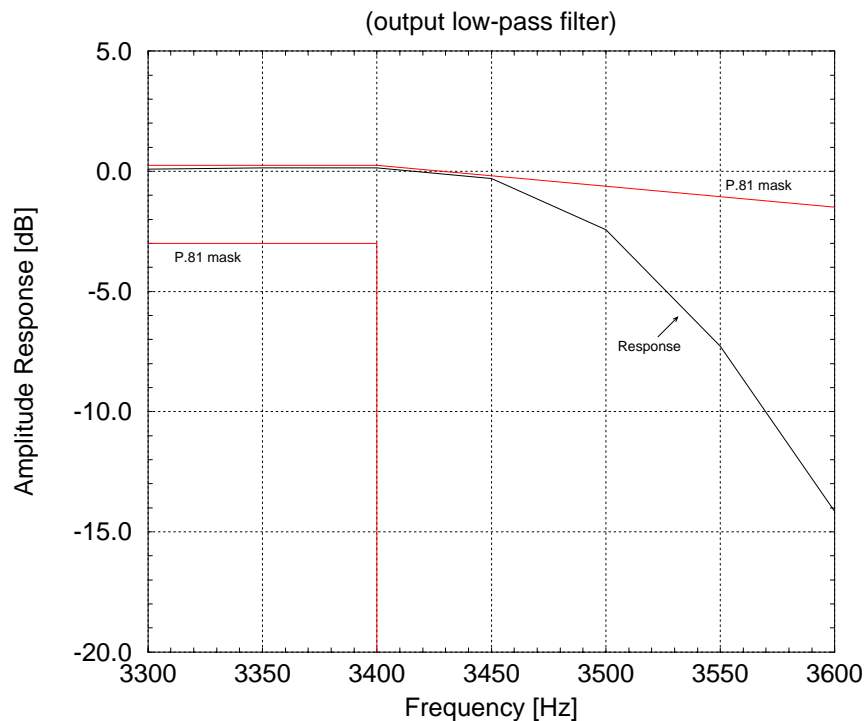
(a) Narrowband Duo-MNRU



(b) Wideband Duo-MNRU

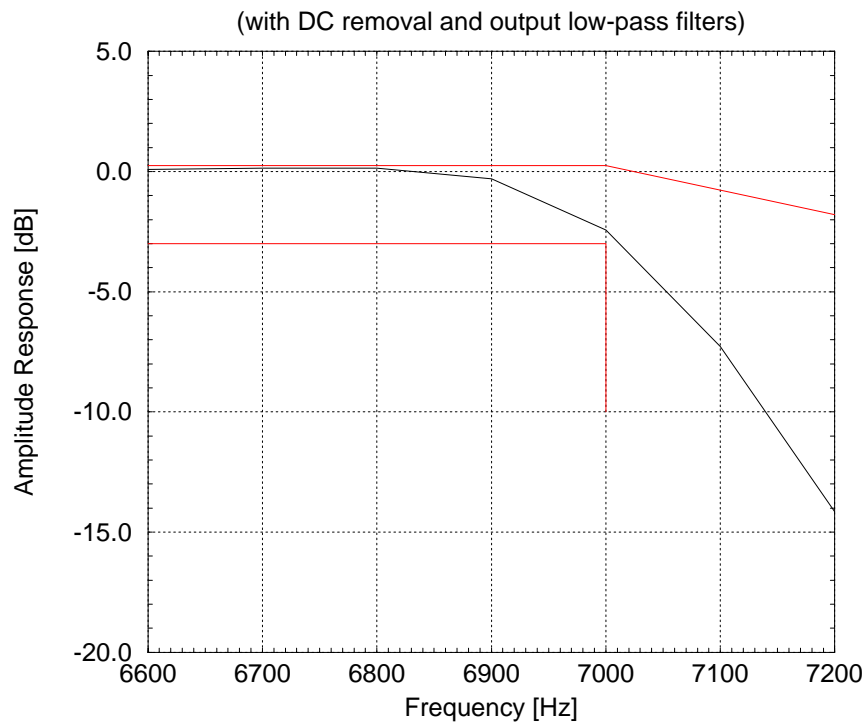
Figure 10.4: DC removal filter for the Duo-MNRU.

Duo-MNRU for narrow-band signals



(a) Narrowband Duo-MNRU

Duo-MNRU response for wideband signals



(b) Wideband Duo-MNRU

Figure 10.5: Output low-pass filter for the Duo-MNRU.

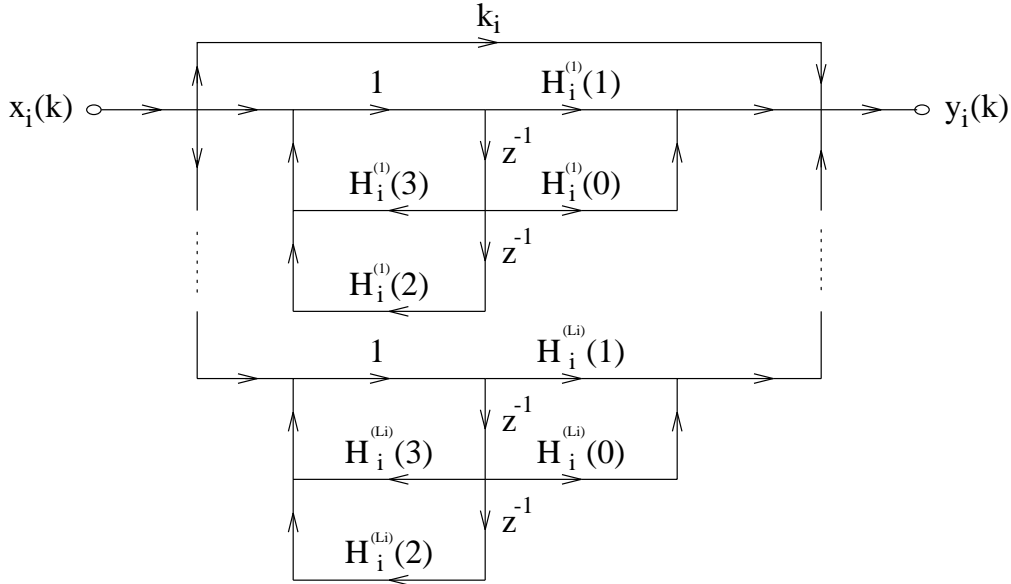


Figure 10.6: MNRU Filters Structure.

wideband case.

The output low-pass filter was implemented using a second-order cascade-form IIR filter with two-sections as illustrated in figure 10.6 and defined by the equation:

$$H_i(z) = A \prod_{k=1}^2 \frac{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2}}{1 + b_{1k}z^{-1} + b_{2k}z^{-2}}$$

IIR filters were chosen because of their low computational complexity when compared to FIR implementations, allowing for a more efficient MNRU implementation.

Random Number Generator for the MNRU module

The random number generator (RNG) used in this implementation was chosen using the following criteria:

- the desired value for Q , Q_d , and the measured Q , Q_m , should be very close for a wide range of Q , e.g., Q from 0 to 50 dB.
- it should show a good approximation of a gaussian distribution. This is needed because it is specified in P.810 and more importantly because uniform distributions do not allow good matching between the desired and measured values of Q .
- the algorithm needed to be portable (i.e., identical results are got in different platforms if the same seed is given).

The RNG chosen to be used in the STL92 version of the MNRU was based on Knuth's Subtractive Method [49],[17, Parts 3.2-3.3], which generates adequate random sequences but is computationally intensive and was too complex to be implemented in a real-time

digital hardware MNRU.

The implementation used in the ITU-T G.729 8 kbit/s speech codec selection tests was based on a gaussian-noise table lookup, in a manner similar to Malden Electronic's MNRU implementation.⁵ This approach is considerably less computationally intensive than the STL92 approach, and was used to further reduce the complexity of the MNRU implementation.

After several experiments [48], a table with 8192 gaussian samples was chosen to be used, which is randomly and uniformly accessed 8 times (i.e., an eight-time sample accumulation) to be used by the MNRU algorithm. The gaussian table itself is generated in run-time (rather than being stored in the data memory of the source or object code) using the Monte-Carlo substitution algorithm. The Monte-Carlo algorithm uses a linear congruential generation (LCG) algorithm defined by

$$I_j = 69069I_{j-1} + 1 \pmod{2^{32}}$$

which is converted to numbers in the range [0..1] using the upper 24 bits of the 32-bit unsigned long I_j . I_0 is a fixed seed equal to 314159265. This algorithm is used to generate the necessary initial random samples for the substitution algorithm.

Once the table has been filled, during the normal operation of the MNRU, eight successive samples are drawn (uniformly) from the table using a different LCG algorithm

$$L_j = 253L_{j-1} + 1 \pmod{2^{24}}$$

of which the upper 13 bits are used to generate random numbers uniformly distributed between 0 and 8191. L_0 is a fixed seed equal to 12345. Both LCGs were implemented as in Aachen University's MNRU implementation.

Since different ranges are necessary for table filling and for gaussian sample generation, two different LCG random number generators were used to avoid any additional calculations due to range conversion and to reduce the software load.

Since the Monte-Carlo RNG is used only at startup time, it is not necessary to keep any state variables for it. The sample-drawing RNG however needs to keep stored the previously generated index, which is stored in a structure of type `RANDOM_state`, whose only field is (as defined in `mnru.h`)⁶:

```
gauss           Index for next random number;
```

The field in `RANDOM_state` should not be altered by the user in any situation.

The operational modes are defined in `mnru.h`:

```
#define RANDOM_RUN 0
#define RANDOM_RESET 1
```

⁵Malden's MNRU uses a ROM table derived from a Gaussian distribution with 4096 samples uniformly distributed throughout the table. An address in the table is uniformly sampled four times and accumulated to form a gaussian noise sample.

⁶The use of a structure instead of a single variable in the parent structure (`MNRU_state`) allows for unimplemented features to be easily added in a later version of the algorithm.

The noise modulation routine is `MNRU_process`, which is described next.

10.2.1 MNRU_process

Syntax:

```
#include "mnru.h"
double *MNRU_process (char operation, MNRU_state *s, float *input, float
                      *output, long n, long seed, char mode, double Q);
```

Prototype: `mnru.h`

Description:

Module for addition of modulated noise to a vector of n samples, according to ITU-T Recommendation P.810, for either the narrowband or the wideband model. Depending on the *mode*, this function:

- adds modulated noise to the *input* buffer at a SNR level of Q dB, saving to *output* buffer (*mode*==MOD_NOISE);
- puts into *output* only the noise, without addition of the original signal (*mode*==NOISE_ONLY);
- produces in the *output* a filtered-only (no noise added) version of the ‘input’ samples (*mode*==SIGNAL_ONLY);

The symbols MOD_NOISE, NOISE_ONLY, and SIGNAL_ONLY are defined in `mnru.h`.

Although the MNRU algorithm operates on a sample-by-sample basis, `MNRU_process` handles the input data in blocks of n samples, for better computational efficiency.

The implementation of the MNRU algorithm has three operational states, called `MNRU_START`, `MNRU_CONTINUE` and `MNRU_STOP`. With `MNRU_START`, the state variables are set, as well as memory is allocated for the intermediate data, and this needs to be the first operation with the algorithm. Differently from the speech voltmeter module, after the initialization of the state variables, the normal calculations are carried out for the first block of data. Once reset, the algorithm changes the operation state to `MNRU_CONTINUE`, and the next calls to the MNRU algorithm will skip the reset operation. With the last block, it is advisable to release the memory allocated to the intermediate data. This is accomplished by calling `MNRU_process` with the operational state set as `MNRU_STOP`. These three operational states are defined in `mnru.h` as follows:

```
#define MNRU_START 1
#define MNRU_CONTINUE 0
#define MNRU_STOP -1
```

Variables:

<i>operation</i>	One of the defined operation status: <code>MNRU_START</code> , <code>MNRU_STOP</code> , <code>MNRU_CONTINUE</code> .
<i>s</i>	A pointer to a <code>MNRU_state</code> structure.

<i>input</i>	Pointer to input float-data vector; must represent 8 or 16 kHz speech samples.
<i>output</i>	Pointer to output float-data vector; will represent 8 or 16 kHz speech samples.
<i>n</i>	Long with the number of samples (float) in input.
<i>seed</i>	Initial value for random number generator.
<i>mode</i>	Operation mode: MOD_NOISE , SIGNAL_ONLY , NOISE_ONLY (description above).
<i>Q</i>	Double defining the desired value for the signal-to-modulated-noise <i>Q</i> for the output data.

Please note that new values of *seed*, *mode*, and *Q* are considered only when *operation* is **MNRU_START**, because they are considered as INITIAL state values. Therefore, when the operation is not **MNRU_START**, they are ignored.

Return value:

Returns a (**double ***)NULL if not initialized or if initialization failed; returns a (**double ***) to an intermediate data vector if reset was successful or is in the **MNRU_CONTINUE** (“run”) operation state.

10.3 Portability and compliance

In the development of this module, several steps were taken to assure its compliance to ITU-T Recommendation P.810, which included:

- agreement of expected and measured *Q* values for tones and speech,
- addition of partial files,
- level of output files,
- frequency response of built-in filters.

Additionally to these objective measurements, a subjective test was performed. The results of this test are found in [48], where it was concluded that the new MNRU implementation conforms to the P.810 and also behaves more closely to the hardware MNRU than the previous STL92 version.

Additionally to the conformance tests, the algorithm was tested for portability using a 1kHz tone file as input to the algorithm with *Q* values ranging from 0 to 50 dB in 5 dB steps, and also for the algorithm in the **SIGNAL_ONLY** mode. The processed test files were then compared the the reference processed files (generated on a HP workstation). Test and reference files should be identical. The algorithm was found to compile and execute correctly on MS-DOS under Borland Turbo-C++ 1.0 and under the MS-DOS port of the GNU-C compiler (gcc), on a HP UNIX workstation with cc (non-ANSI) and gcc, on a Sun workstation with cc (non-ANSI) and also on VAX VMS and APX computers.

10.4 Example code

10.4.1 Description of the demonstration programs

One demonstration program is provided for the MNRU module, `mnrudemo.c`. Irrespective of whether the 16-bit, linear PCM input file is sampled at 8 or 16 kHz, program `mnrudemo.c` will add the multiplicative noise signal to the input signal at the user-defined Q level and produce as output a 16-bit, linear PCM file. Optionally, the program can produce a signal-only file (equivalent to a very high Q value) or a noise-only file (the signal path is disconnected).

10.4.2 Simple example

The following C code gives an example of a possible use of the Duo-MNRU module. The input file `speech` is added to a multiplicative noise at a SNR defined by parameter Q . All samples in the file are processed.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "ugstdemo.h"
#include "mnruc.c"           /* ... Include MNRU module ... */
#include "ugst-utl.c"        /* ... Include of utilities ... */

#define BLK_LEN 256
main(argc, argv)
    int      argc;
    char     *argv[];
{
    /* File variables */
    char      FileIn[80], FileOut[80];
    FILE      *Fi, *Fo;
    MNRU_state state;
    short     Buf[BLK_LEN];
    float     inp[BLK_LEN], out[BLK_LEN];
    double     QdB;
    long      l;
    char      MNRU_mode = MOD_NOISE, operation;

    /* Read parameters for processing */
    GET_PAR_S(1, "_Input File: ..... ", FileIn);
    GET_PAR_S(2, "_Output File: ..... ", FileOut);
    GET_PAR_D(3, "_Desired Q: ..... ", QdB);

    /* Check for parameter 4 to change MNRU operation mode */
    if (argc > 4)
    {
        MNRU_mode = toupper(argv[5][0]);
    }
}
```

```

    if (MNRU_mode == 'S')          /* Signal-only mode */
        MNRU_mode = SIGNAL_ONLY;
    else if (MNRU_mode == 'M')     /* Modulated noise, the default mode */
        MNRU_mode = MOD_NOISE;
    else if (MNRU_mode == 'N')     /* Noise-only mode */
        MNRU_mode = NOISE_ONLY;
    else
    {
        fprintf(stderr, "Bad mode chosen; use M,N,or S \n"); exit(2);
    }
}

/* Opening input and output files */
Fi = fopen(FileIn, RB);
Fo = fopen(FileOut, WB);

/* INSERTION OF MODULATED NOISE ACCORDING TO P.810 (FEB.96) */

/* Set operation as start */
operation = MNRU_START;

/* Process for all samples in file */
while ((l = fread(Buf, sizeof(short), BLK_LEN, Fi)) != NULL)
{
    /* Convert data from 16-bit short to normalized float */
    sh2fl_16bit((long) l, Buf, inp, 1);

    /* MNRU processing */
    MNRU_process(operation, &state, inp, out, l, 314159265L, MNRU_mode, QdB);

    /* Change operation mode: START --> CONTINUE */
    if (operation == MNRU_START)
        operation = MNRU_CONTINUE;

    /* Convert from normalized float to short (hard clip and rounding) */
    fl2sh_16bit((long) l, out, Buf, 1);

    /* Save data to file */
    fwrite(Buf, sizeof(short), l, Fo);
}

/* Stop mode: Deallocation of memory, but process 0 samples */
operation = MNRU_STOP;
MNRU_process (operation, &state, inp, out, 0L, 0L, 0, (double) 0.0);

/* Finalizations */
fclose(Fi);
fclose(Fo);
return(0);
}

```

Chapter 11

SVP56: The Speech Voltmeter

11.1 Description of the Algorithm

The specification for the measurement of the active level of speech signals is given in ITU-T Recommendation P.56 [50], and is commonly referred as *speech voltmeter*¹. Besides the description above, there is complementary information in the ITU-T *Handbook on Telephony* [51], section on ‘Measurement of Speech’.

In summary, the P.56 algorithm takes samples of a signal in the speech bandwidth and calculates its active speech level. This means that silence and idle noise are not taken into account when calculating the level of the signal. Furthermore, *structural pauses* (pauses in the range of 250 ms which are inherent to the utterance process) are considered in the measurements, but *grammatical pauses* (pauses between phrases or to emphasise words, generally in the range of 300 ms or more) are excluded, because they do not contribute to speech subjective loudness [51].

To decide about the activity or inactivity of a speech segment, the algorithm calculates an envelope waveform, or short-term mean amplitude, such that pauses shorter than 100 ms are not excluded, but pauses longer than 350 ms are². A signal is considered *active* when its short-term mean level (envelope) exceeds a threshold level (or *margin* of) 15.9 dB below the prevailing speech voltage³, and also during short gaps between such bursts of activity.

A word of caution must be given here: the above mentioned margin above of 15.9 dB has been optimized for speech with a low level of background noise. This means that in the case of generation of material for listening subjective tests, once the original files have been processed (already level equalized), especially by processes that add significant amount of noise to the files (e.g. MNRU for low values of Q), the P.56 algorithm shall not be utilized for re-equalization. Since the noise introduced by the processing algorithm will be far above the threshold discussed, the P.56 algorithm will generate wrong measurements of the active level and speech activity. A practical way to observe whether the P.56 may

¹After the British Telecom and Malden Ltd.’s SV6 Speech Voltmeter.

²Users will perceive a pause when it lasts more than about 350 ms.

³The margin of 15.9 dB has been chosen to be comfortably above the circuit noise, while causing few false detections or failures to detect, having being determined by subjective experiments.

be utilized on processed files is to observe the activity factor: if it increases significantly in relation to the original file's activity factor, then the use of the P.56 for re-equalizations should be discarded.

Another operating assumption for the P.56 Recommendation suggests that the input signal be band-limited (300–3400 Hz for telephony band signals and 100–7000 Hz for wideband signals), as given in Table 3 and Figure 2 of P.56.

The speech voltmeter algorithm is expressed in terms of discrete operations. Because of this, a minimum sampling frequency must be chosen, and the specification in P.56 gives it as 600 Hz. This is well below Nyquist frequency of the digitized sample's normally used for telephony applications, either 8000 or 16000 Hz, which is explained by the fact that the matter of interest here is not signal's frequency content's information, but only signal statistics. This is one of the unspecified details of the P.56 that may cause implementations to differ.

After considering an input sample x_i , the speech voltmeter performs two operations. First, the total energy of the signal is calculated (sq), updating also the number of samples n and signal's (long-term) mean level s . Second, the envelope (or short-term mean level) q of the signal is extracted using a second-order exponential filtering:

$$p_i = g \cdot p_{i-1} + (1 - g) \cdot |x_i|$$

$$q_i = g \cdot q_{i-1} + (1 - g) \cdot p_i$$

with initial states $p_0 = q_0 = 0$ and the quantity g defined as:

$$g = \exp(-1/(f \cdot T))$$

for f as the sampling frequency, in Hz, and T , a time constant for smoothing, equal to 0.030s (30 ms).

With the envelope calculated, the algorithm calculates the number of times that the envelope exceeds each of the threshold levels. The thresholds are represented in a vector c of $B - 1$ positions, where B is the resolution (number of bits) of the samples. The values in this vector range from half the maximum possible amplitude down to (or less than) one LSB (Least Significant Bit). In terms of practical implementations, the values of c_j are a power of 2:

$$c_j = 2^j, \quad j = 0 \cdots B - 2$$

There are three possible cases⁴:

- **the envelope exceeds the threshold c_j :** increment the *activity counter* for the quantization level j , a_j , and set the timer vector (or *hangover counter*) h_j to zero. This operation means that the segment is active as far as the level j is concerned, and so the hangover counter must be set to zero, as well as the number of active samples (a_j) incremented.

⁴It is interesting to remark that the lower the threshold level, the greater the activity count for that level will be.

- **the envelope does not exceed the threshold level, but the hangover counter h_j is less (or shorter) than I samples:** this means that, besides the sample being into a pause segment (because the level is below the threshold), it is a structural pause (because the time spent since the last activity burst is less than 200ms). Therefore, the action here is to increment the activity counter, as well as the hangover counter for the level j .
- **the envelope does not exceed the threshold level and the hangover time exceeds I samples:** this means that the sample is into a pause (because the level is below the threshold); moreover, it is a grammatical pause (because the time spent since the last activity burst is more than 200ms). Therefore, no increments are done.

Then, after all the samples of interest have been considered, three quantities have been accumulated:

1. total number of samples, n ;
2. signal energy, sq ;
3. an activity count a_j for each threshold level c_j , $j = 0 \cdots B - 2$.

The active level can be evaluated from these three parameters, as follows. First, the long-term mean level is calculated:

$$L = 10 \log_{10}(sq/n) - 20 \log(r)$$

and the activity counter and threshold vectors are converted to dB:

$$A_j = 10 \log_{10}(sq/a_j) - 20 \log(r)$$

$$C_j = 20 \log_{10}(c_j) - 20 \log(r)$$

where r is the 0 dB reference point for the measurements⁵.

In sequence, the difference between A_j and C_j is calculated for each j . When this difference lies below the margin M (15.9 dB), then the active level⁶ A is found by interpolating between this level \hat{j} and level $\hat{j} - 1$ (i.e., the nearest level k where $A_k - C_k > M$, what gives $k = \hat{j} - 1$), using a bipartition (binary) interpolation algorithm. There are three special cases here:

- When $\hat{j} = 0$, then the active level is zero;
- When $|A_{\hat{j}} - C_{\hat{j}} - M| \leq \delta$ (where δ is the the given tolerance, or degree of accuracy): the active level is $A_{\hat{j}}$.
- When $|A_{\hat{j}-1} - C_{\hat{j}-1} - M| \leq \delta$: the active level is $A_{\hat{j}-1}$.

The tolerance δ is not specified in P.56, hence being implementation-dependent.

Once the active level is found, the only remaining point is the calculation of the activity factor,

$$Activity = 10^{L-A}$$

or, in percents,

$$Activity_{\%} = 100 \cdot 10^{L-A}$$

⁵This is another unspecified detail in P.56. This implementation's choice is given in next section.

⁶The true active level is defined as the one which exceeds the threshold used for its derivation by a $M = 15.9$ dB.

11.2 Implementation

This implementation of the speech voltmeter algorithm can be found in the module `sv-p56.c`, with prototypes in `sv-p56.h`. This version evolved from a preliminary Fortran implementation provided by Telebrás, Brazil, which was used by several laboratories, in especial by participants of ETSI's contest for the second generation of Digital Mobile Radio Systems.

In Recommendation P.56, there are several undefined issues needed to be resolved for the implementation of this module. Especially, the rate f used for the averages and the tolerance, or degree of accuracy, δ to be used for the interpolation of the active level have to be defined. Another undefined parameter is the reference level, or 0 dB reference point r . The choices of this implementation are shown in the table below:

Speech voltmeter parameters		
Parameter	Description	Value
f	sampling rate	same rate of the input signal.
r	dB reference	0 dBov (see Chapter 2).
δ	tolerance	± 0.5 dB (the same of M).

The P.56 algorithm operates on a sample-by-sample basis. However, since most software implementations use blocks (or frames) of samples, the speech voltmeter was designed to work with blocks of samples. Measurements are cumulative, therefore state variables are needed in this approach. These state variables have been arranged as fields of a structure whose name is `SVP56_state`. The fields of the structure are⁷:

f	Sampling frequency, in Hz
$a[15]$	Activity count
$c[15]$	Threshold level
$hang[15]$	Hangover count
n	Number of samples read since last reset
s	Sum of all samples since last reset
sq	Squared sum of samples since last reset
p	Intermediate quantities
q	Envelope
max	Max absolute value found since last reset
$refdB$	0 dB reference point, in [dB]
$rmsdB$	RMS value found since last reset
$maxP$	Most positive value since last reset
$maxN$	Most negative value since last reset
$DClevel$	Average level since last reset
$ActivityFactor$	Activity factor since last reset

The user should note that although some fields are of interest to report signal statistics, such as long-term level, extreme values for file, average (or DC) level, etc., these values

⁷All the fields are `double`, except the `float` f and the `unsigned long` $a[]$, $hang[]$, and n .

shall not be altered. See section 11.2.3, which describes macros for safe inspection of the parameters of interest.

The algorithm has two operational parts, one that deals with the initialization of the state variables, and is carried out by the function `init_speech_voltmeter`, and the measuring part (or the algorithm itself), carried out by `speech_voltmeter`. These are presented in the next two sections.

11.2.1 `init_speech_voltmeter`

Syntax:

```
#include "sv-p56.h"
void init_speech_voltmeter (SVP56_state *state, double f);
```

Prototype: sv-p56.h

Description:

`init_speech_voltmeter` performs the initialization of the speech voltmeter state variables in the structure pointed by *state* to the appropriate initial values. The only value required from the user is the sampling rate *f* (in Hz) of the signal that the speech voltmeter is supposed to measure. Note that when measuring new speech material, the state variable shall be re-initialized, otherwise accumulation of previous measurements will happen and wrong measurements will be reported.

Variables:

<i>state</i>	Is a pointer to a speech voltmeter state variable.
<i>f</i>	Is the sampling rate (in Hz) of the signal to be measured in the next calls of <code>speech_voltmeter</code> . If zero or negative, the sampling rate is initialized to 16000 Hz.

Return value:

None.

11.2.2 `speech_voltmeter`

Syntax:

```
#include "sv-p56.h"
double speech_voltmeter (float *buffer, long smpno, SVP56_state *state);
```

Prototype: sv-p56.h

Description:

`speech_voltmeter` performs the measurement of the active level of a speech signal according to ITU-T Recommendation P.56. Other relevant statistics are also available in the state variable (for details, see section 11.2.3 ahead):

- average level;
- maximum and minimum amplitude values;
- rms power, in dB;

Variables:

buffer Is the input sample `float` buffer.
smpno Is the number of samples in *buffer*.
state Is a pointer to the state variable buffer. This shall have been initialized by a previous call to `init_speech_voltmeter`.

Return value: Returns the active speech level, in dB relative to dBov, as a double.

11.2.3 Getting state variable fields

Some macros are provided for the inspection of the speech voltmeter statistics:

Syntax:

```
#include "sv-p56.h"
SVP56_get_rms_dB(SVP56_state state);
SVP56_get_DC_level(SVP56_state state);
SVP56_get_activity(SVP56_state state);
SVP56_get_pos_max(SVP56_state state);
SVP56_get_neg_max(SVP56_state state);
SVP56_get_abs_max(SVP56_state state);
SVP56_get_smpno(SVP56_state state);
```

Description:

`SVP56_get_rms_dB` and `SVP56_get_DC_level` return respectively the long-term level (in dBov) and the DC level (in the normalized range) calculated for the material, both as a double.

`SVP56_get_activity` returns the activity factor as a double, in percents (0..100%).

`SVP56_get_pos_max`, `SVP56_get_neg_max`, and `SVP56_get_abs_max` returns respectively the maximum positive, negative and absolute amplitudes found for the input data, as normalized double values (range -1.0..+1.0).

`SVP56_get_smpno` returns as a `unsigned long` the total number of samples.

Variables:

All the macros expect a valid SVP56 state variable structure (not a pointer!).

11.3 Portability and compliance

Compliance tests of this module have been done based on the compliance with other existing implementations, especially of the Deutsches Bundespost Telekom Forschungs Institute. Reported results were found to be within the error margins of the P.56 algorithm.

Portability was checked by running the same speech file on a proven platform and on a test platform. Results have to be identical, in especial long-term and active levels, as well as the activity factor. During the development of this tool, the provided demonstration programs (see section 11.4) were used to measure and level-equalize a reference file. These test files are provided in the STL distribution.

This module had portability tested for VAX/VMS with VAX-C and GNU C (gcc) and for MS-DOS with a number of Borland C/C++ compilers (Turbo C v2.0, Turbo-C++ v1.0, Borland C++ v3.1). Portability was also tested in a number of Unix workstations and compilers: Sun workstation with Sun-OS and Sun-C (cc), acc, and gcc; HP workstation with HP-UX and gcc.

11.4 Examples

11.4.1 Description of the demonstration programs

As a part of the speech voltmeter module, two example programs are provided. They are called `sv56demo.c` and `actlevel.c`.

Both example programs calculate the equalization factor to equalize the active speech level of a file 'NdB' dBs below the 0 dBov reference using the algorithm described in this chapter. However, only program `sv56demo.c` carry out the level-equalization of the input file, which is saved in an aoutput file. Levels are reported in dBov.

In general, input files are in integer representation, 16-bit words, 2's complement (i.e., **short** data). In UGST convention, this data must be left-adjusted, *rather* than right-adjusted. Since the speech voltmeter uses **float** input data, it is necessary to convert from **short** (in the mentioned format) to **float**; this is carried out by the function `sh2fl()`. In addition, the option to 'normalize' the input data to the range -1..+1 is selected. After the equalization factor is found, results are reported on the screen, which varies according to the program used and some of the command-line options.

While program `actlevel.c` stops at this point, program `sv56demo.c` proceeds calling the function `scale()` to carry out the (amplitude) equalization using single (rather than double) float precision. After equalization, the samples are converted back to integer (short, right-justified) with the routine `fl2sh()` using truncation, no zero-padding of the least significant bits, left-justification of data, and hard-clipping of data above the overload point. After that, data is saved to the user-specified file .

11.4.2 Small example

Following is an simplification of the described demonstration programs. It only measures the statistics for the input file, without carrying out level equalizations and does not implement the several command-line options of `actlevel.c`.

```
#include <stdio.h>
#include <stdlib.h>
```

```

#include <math.h>
#include "ugstdemo.h"          /* ... UGST demonstration program defs ... */
#include "sv-p56.h"           /* ... SV-P56 prototypes & defs ... */
#include "ugst-utl.h"         /* ... UGST utilities ... */
#define BLK_LEN 256

void main(argc, argv)
    int      argc;
    char     *argv[];
{
    SVP56_state  state;      /* Speech voltmeter state */
    char         FileIn[180]; /* input file name */
    FILE         *Fi;        /* input file pointers */
    long         N=BLK_LEN, 1;
    short        bitno, buffer[BLK_LEN];
    float        Buf[BLK_LEN];
    double       ActiveLeveldB, sf, satur;

    /* Reads parameters for processing */
    GET_PAR_S(1, "_Input File: ..... ", FileIn);

    /* Checks parameters 2, and 3 for specification in command line */
    FIND_PAR_D(2, "_Sampling Frequency: .. ", sf, 16000);
    FIND_PAR_L(3, "_A/D resolution: ..... ", bitno, 16);

    /* Calculate overload point in the non-normalized range */
    satur = pow ((double)2.0, (double)(bitno - 1));

    /* Reset- variables for speech level measurements */
    init_speech_voltmeter(&state, sf);

    /* Opening input file */
    Fi = fopen(FileIn, RB);

    /* Read samples ... */
    while ((l = fread(buffer, N, sizeof(short), Fi)) > 0)
    {
        /* ... Convert samples to float, normalizing to +1..-1 */
        sh2fl((long) l, buffer, Buf, (long) state.bitno, 1);

        /* ... Get the active level */
        ActiveLeveldB = speech_voltmeter(Buf, (long) l, &state);
    }

    /* If the activity factor is 0, don't report many things */
    if (SVP56_get_activity(state) == 0)
        printf("\n Activity factor is ZERO -- the file is silence or idle noise");
    else
    {
        printf("\n DC level: ..... %7.0f [PCM]",

```

```
        SVP56_get_DC_level(state) * satur);
    printf("\n  Maximum positive value: .. %7.0f [PCM]",
        SVP56_get_pos_max(state) * satur);
    printf("\n  Maximum negative value: .. %7.0f [PCM]",
        SVP56_get_neg_max(state) * satur);
    printf("\n  Long-term energy (rms): .. %7.3f [dBov]",
        SVP56_get_rms_dB(state);
    printf("\n  Active speech level: ..... %7.3f [dBov]", ActiveLeveldB);
    printf("\n  Activity factor: ..... %7.3f [%%]",
        SVP56_get_activity(state));
}
fclose(Fi);
}
```


Chapter 12

ITU-T Basic Operators

12.1 Overview of basic operator libraries

The fixed-point descriptions of G.723.1 and G.729 are based on 16 and 32-bit arithmetic operations defined by ETSI in 1993 for the standardisation of the half-rate GSM speech codec. These operations are also used to define the GSM enhanced full-rate (EFR) and adaptive multi-rate (AMR) speech codecs. [1]

Although almost identical to the G.729 basic operators library, the ETSI EFR/AMR library contains a number of modifications that should be incorporated into an STL version of the library. These include trapping excessive shift values and a modification to the generation of the carry flag (which is not actually used in G.729).

12.2 Description of the 32-bit basic operators and associated weights

The following describes the different basic operators available in the STL, and are organized by complexity (“weights”). The reported complexity values were current as of the publication of the STL2000, however a revision is being considered by UGST.

12.2.1 Variable definitions

The variables used in the operators are signed integers in 2’s complements representation, defined by:

`v1`, `v2`: 16 bit variables

`L_v1`, `L_v2`, `L_v3`: 32 bit variables

12.2.2 Arithmetic operators with complexity weight of 1

add(v1, v2)

Performs the addition ($v1+v2$) with overflow control and saturation; the 16 bit result is set at +32767 when overflow occurs or at -32768 when underflow occurs.

sub(v1, v2)

Performs the subtraction ($v1-v2$) with overflow control and saturation; the 16-bit result is set at +32767 when overflow occurs or at -32768 when underflow occurs.

abs_s(v1)

Absolute value of v1. If v1 is -32768, returns 32767.

shl(v1, v2)

Arithmetically shift the 16 bit input v1 left v2 positions. Zero fill the v2 LSB of the result. If v2 is negative, arithmetically shift v1 right by -v2 with sign extension. Saturate the result in case of underflows or overflows.

shr(v1, v2)

Arithmetically shift the 16 bit input v1 right v2 positions with sign extension. If v2 is negative, arithmetically shift v1 left by -v2 and zero fill the -v2 LSB of the result:

$\text{shr}(v1, v2) = \text{shl}(v1, -v2)$

Saturate the result in case of underflows or overflows.

extract_h(L_v1)

Return the 16 MSB of L_v1.

extract_l(L_v1)

Return the 16 LSB of L_v1.

mult(v1, v2)

Performs the multiplication of v1 by v2 and gives a 16 bit result which is scaled, i.e.

$\text{mult}(v1, v2) = \text{extract_l}(\text{L_shr}((v1 \text{ times } v2), 15))$

Note that $\text{mult}(-32768, -32768) = 32767$.

L_mult(v1, v2)

Operator L_mult implements the 32 bit result of the multiplication of v1 times v2 with one shift left, i.e.

$\text{L_mult}(v1, v2) = \text{L_shl}((v1 \times v2), 1)$

Note that `L_mult(-32768, -32768) = 2147483647`.

`L_mult0(v1, v2)`

Operator `L_mult0` implements the 32 bit result of the multiplication of `v1` times `v2` *without* left shift, i.e.

$$\text{L_mult}(v1, v2) = (v1 \times v2)$$

`negate(v1)`

Negate `v1` with saturation, saturate in the case when input is `-32768`:

$$\text{negate}(v1) = \text{sub}(0, v1)$$

`round(L_v1)`

Round the lower 16 bits of the 32 bit input number into the most significant 16 bits with saturation. Shift the resulting bits right by 16 and return the 16 bit number:

$$\text{round}(L_v1) = \text{extract_h}(\text{L_add}(L_v1, 32768))$$

`L_mac(L_v3, v1, v2)`

Multiply `v1` by `v2` and shift the result left by 1. Add the 32 bit result to `L_v3` with saturation, return a 32 bit result:

$$\text{L_mac}(L_v3, v1, v2) = \text{L_add}(L_v3, \text{L_mult}(v1, v2))$$

`L_msu(L_v3, v1, v2)`

Multiply `v1` by `v2` and shift the result left by 1. Subtract the 32 bit result from `L_v3` with saturation, return a 32 bit result:

$$\text{L_msu}(L_v3, v1, v2) = \text{L_sub}(L_v3, \text{L_mult}(v1, v2)).$$

`L_mac0(L_v3, v1, v2)`

Multiply `v1` by `v2` *without* left shift. Add the 32 bit result to `L_v3` with saturation, returning a 32 bit result:

$$\text{L_mac}(L_v3, v1, v2) = \text{L_add}(L_v3, \text{L_mult0}(v1, v2))$$

`L_msu0(L_v3, v1, v2)`

Multiply `v1` by `v2` *without* left shift. Subtract the 32 bit result from `L_v3` with saturation, returning a 32 bit result:

$$\text{L_msu}(L_v3, v1, v2) = \text{L_sub}(L_v3, \text{L_mult0}(v1, v2)).$$

`L_macNs(L_v3, v1, v2)`

Multiply `v1` by `v2` and shift the result left by 1. Add the 32 bit result to `L_v3` without saturation, return a 32 bit result. Generates carry and overflow values:

`L_macNs(L_v3, v1, v2) = L_add_c(L_v3, L_mult(v1, v2))`

`L_msuNs(L_v3, v1, v2)`

Multiply `v1` by `v2` and shift the result left by 1. Subtract the 32 bit result from `L_v3` without saturation, return a 32 bit result. Generates carry and overflow values:

`L_msuNs(L_v3, v1, v2) = L_sub_c(L_v3, L_mult(v1, v2))`

`L_mls(L_v1, v2)`

Performs a multiplication of a 32-bit variable `L_v1` by a 16-bit variable `v2`, returning a 32-bit value.

`div_l(L_v1, v2)`

Produces a result which is the fractional integer division of a positive 32-bit value `L_v1` by a positive 16-bit value `v2`. The result is positive (leading bit equal to 0) and truncated to 16 bits.

`i_mult(v1, v2)`

Multiply two 16-bit words `v1` and `v2` returning a 16-bit word with overflow control.

12.2.3 Arithmetic operations with complexity weight of 2

`L_add(L_v1, L_v2)`

This operator implements 32 bit addition of the two 32 bit variables (`L_v1+L_v2`) with overflow control and saturation; the result is set at +2147483647 when overflow occurs or at -2147483648 when underflow occurs.

`L_sub(L_v1, L_v2)`

32 bit subtraction of the two 32 bit variables (`L_v1-L_v2`) with overflow control and saturation; the result is set at +2147483647 when overflow occurs or at -2147483648 when underflow occurs.

`L_add_c(L_v1, L_v2)`

Performs the 32 bit addition with carry. No saturation. Generates carry and overflow values. The carry and overflow values are binary variables which can be tested and assigned values.

`L_sub_c(L_v1, L_v2)`

Performs the 32 bit subtraction with carry (borrow). Generates carry (borrow) and overflow values. No saturation. The carry and overflow values are binary variables which can be tested and assigned values.

L_negate(L_v1)

Negate the 32 bit L_v1 with saturation, saturate in the case where input is -2147483648.

L_shl(L_v1, v2)

Arithmetically shift the 32 bit input L_v1 left v2 positions. Zero fill the v2 LSB of the result. If v2 is negative, arithmetically shift L_v1 right by -v2 with sign extension. Saturate the result in case of underflows or overflows.

L_shr(L_v1, v2)

Arithmetically shift the 32 bit input L_v1 right v2 positions with sign extension. If v2 is negative, arithmetically shift L_v1 left by -v2 and zero fill the -v2 LSB of the result. Saturate the result in case of underflows or overflows.

mult_r(v1, v2)

Same as mult() but with rounding, i.e.

```
mult_r(v1, v2) = extract_l(L_shr(((v1 × v2)+16384), 15))
```

and mult_r(-32768, -32768) = 32767.

shr_r(v1, v2)

Same as shr() but with rounding. Saturate the result in case of underflows or overflows.

```
if (v2>0) then
  if (sub(shl(shr(v1,v2),1), shr(v1,sub(v2,1)))=0)
  then shr_r(v1, v2) = shr(v1, v2)
  else shr_r(v1, v2) = add(shr(v1, v2), 1)
else if (v2 ≤ 0)
  then shr_r(v1, v2) = shr(v1, v2)
```

shift_r(v1, v2)

Same as shl() but with rounding. Saturate the result in case of underflows or overflows:

```
shift_r(v1, v2) = shr_r(v1, -v2)
```

mac_r(L_v3, v1, v2)

Multiply v1 by v2 and shift the result left by 1. Add the 32 bit result to L_v3 with saturation. Round the 16 least significant bits of the result into the 16 most significant bits with saturation and shift the result right by 16. Returns a 16 bit result.

```
mac_r(L_v3, v1, v2) =
  round(L_mac(L_v3, v1, v2))=
  extract_h(L_add(L_add(L_v3, L_mult(v1, v2)), 32768))
```

`msu_r(L_v3, v1, v2)`

Multiply `v1` by `v2` and shift the result left by 1. Subtract the 32 bit result from `L_v3` with saturation. Round the 16 least significant bits of the result into the 16 bits with saturation and shift the result right by 16. Returns a 16 bit result.

```
msu_r(L_v3, v1, v2) =
    round(L_msu(L_v3, v1, v2))=
    extract_h(L_add(L_sub(L_v3, L_mult(v1, v2)), 32768))
```

`L_deposit_h(v1)`

Deposit the 16 bit `v1` into the 16 bit most significant bit of the 32 bit output. The 16 least significant bits of the output are zeroed.

`L_deposit_l(v1)`

Deposit the 16 bit `v1` into the 16 bit least significant bit of the 32 bit output. The 16 most significant bits of the output are sign extended.

12.2.4 Arithmetic operations with complexity weight of 3

`L_shr_r(L_v1, v2)`

Same as `L_shr(v1,v2)` but with rounding. Saturate the result in case of underflows or overflows:

```
if (v2 > 0) then
    if (L_sub(L_shl(L_shr(L_v1,v2),1), L_shr(L_v1, sub(v2,1)))) == 0
    then L_shr_r(L_v1, v2) = L_shr(L_v1, v2)
    else L_shr_r(L_v1, v2) = L_add(L_shr(L_v1, v2), 1)
if (v2 ≤ 0)
    then L_shr_r(L_v1, v2) = L_shr(L_v1, v2)
```

`L_shift_r(L_v1, v2)`

Same as `L_shl(L_v1,v2)` but with rounding. Saturate the result in case of underflows or overflows.

```
L_shift_r(L_v1, v2) = L_shr_r(L_v1, -v2)
```

`L_abs(L_v1)`

Absolute value of `L_v1`, with `L_abs(-2147483648)=2147483647`.

12.2.5 Arithmetic operations with complexity weight of 4

`L_sat(L_v1)`

Long (32 bit) `L_v1` is set to 2147483647 if an overflow occurred, or -2147483648 if an underflow occurred, on the most recent `L_add_c()`, `L_sub_c()`, `L_macNs()` or `L_msuNs()` operations. The carry and overflow values are binary variables which can be tested and assigned values.

12.2.6 Arithmetic operations with complexity weight of 15

`norm_s(v1)`

Produces the number of left shifts needed to normalise the 16 bit variable `v1` for positive values on the interval with minimum of 16384 and maximum 32767, and for negative values on the interval with minimum of -32768 and maximum of -16384; in order to normalise the result, the following operation must be done:

```
norm_v1 = shl(v1, norm_s(v1))
```

12.2.7 Arithmetic operations with complexity weight of 18

`div_s(v1, v2)`

Produces a result which is the fractional integer division of `v1` by `v2`. Values in `v1` and `v2` must be positive and `v2` must be greater than or equal to `v1`. The result is positive (leading bit equal to 0) and truncated to 16 bits. If `v1=v2`, then `div(v1, v2) = 32767`.

12.2.8 Arithmetic operations with complexity weight of 30

`norm_l(L_v1)`

Produces the number of left shifts needed to normalise the 32 bit variable `L_v1` for positive values on the interval with minimum of 1073741824 and maximum 2147483647, and for negative values on the interval with minimum of -2147483648 and maximum of -1073741824; in order to normalise the result, the following operation must be done:

```
L_norm_v1 = L_shl(L_v1, norm_l(L_v1))
```

12.2.9 Complexity associated with data moves, logical operations, arithmetic test and other operations

Data moves

Each data move on **short** (16 bits) data has a weight of 1 and each data move on **long** (32 bits) has a weighted of 2.

A short variable cannot be moved to a long variable directly, and a long variable cannot be moved to a short variable directly. In these cases, functions such as **round()**, **extract_l()**, **extract_h()**, **L_deposit_l()**, **L_deposit_h()** must be used.

There will be no extra weighting for data move for the following functions: **extract_l()**, **extract_h()**, **L_deposit_l()** and **L_deposit_h()** (the weighting of the data move is already included in the weighting of these functions).

Data moves are only counted in the following cases:

1. A data move from a constant to a variable;
2. A data move from a variable to a variable;
3. A data move of the result of a basic operation to an array variable;
4. When an arithmetic test is performed on an array variable.

Logical operations

A logical operation is one of the following: **And**, **Or**, **Xor**, and **Not**. Similarly to the move operation, logical operations involving **short** and **long** data will have a weight of 1 and 2, respectively.

Arithmetic tests

Each arithmetic test (short or long) has a weight of 2.

All arithmetic test on data must be presented as a compare to zero. To perform comparison between two variables (or a variable and a non-zero constant), a subtract (**sub** or **L_sub**) must be performed first.

Other operations

Address computation must be excluded from the complexity evaluation. However, when extremely complex address computations are done, these address computations should be resolved using the basic operations, in order to account for the associated complexity.

There is no complexity counted for any loops, subroutine calls, etc., except for the complexity for arithmetic test on data in program control statement (e.g. **do while**).

Table 12.1 contains a survey of basic operator usage.

Table 12.1: Use of 32-bit basic operators in G.723.1, G.729 and ETSI GSM speech coding recommendations.

Operation	Weight	FR GSM	HR GSM	EFR GSM	AMR GSM	G.729	G.723.1	TETRA
add()	1	X	X	X	X	X	X	X
sub()	1	X	X	X	X	X	X	X
abs_s()	1	X (1)	X	X	X	X	X	X
shl()	1	X (2)	X	X	X	X	X	X
shr()	1	X (3)	X	X	X	X	X	X
extract_h()	1		X	X	X	X	X	X
extract_l()	1	X (4)	X	X	X	X	X	X
mult()	1	X	X	X	X	X	X	X
L_mult()	1	X	X	X	X	X	X	X
negate()	1	X (5)	X	X	X	X	X	
round()	1		X	X	X	X	X	X
L_mac()	1		X	X	X	X	X	X
L_msu()	1		X	X	X	X	X	X
L_macNs()	1			X		X	X	
L_msuNs()	1					X	X	
L_add()	2	X	X	X	X	X	X	X
L_sub()	2	X	X	X	X	X	X	X
L_add_c()	2						X	
L_sub_c()	2						X	
L_negate()	2	X (6)	X	X	X	X	X	X
L_shl()	2	X (7)	X	X	X	X	X	X
L_shr()	2	X (8)	X	X	X	X	X	X
mult_r()	2	X	X	X	X	X	X	X
shr_r()	2		X (12)	X	X	X	X	
mac_r()	2		X				X	
msu_r()	2		X				X	
L_deposit_h()	2		X	X	X	X	X	X
L_deposit_l()	2	X (9)	X	X	X	X	X	X
L_shr_r()	3		X (13)	X	X	X	X	X
L_abs()	3		X	X	X	X	X	X
L_sat()	4					X	X	
norm_s()	15		X	X	X	X	X	
div_s()	18	X (10)	X (14)	X	X	X	X	X
norm_l()	30	X (11)	X	X	X	X	X	X
L_mls()	1						X (15)	
div_l()	1						X (15)	
i_mult()	1						X (15)	
L_mult0()	1							X (16)
L_mac0()	1							X (16)
L_msu0()	1							X (16)

Notes to Table 12.1:

1. `abs_s(v1)` is referred to as `abs(v1)` in GSM 06.10 (GSM full-rate).
2. `shl(v1,v2)` is written as `v1<<v2` in GSM 06.10.
3. `shr(v1,v2)` is written as `v1>>v2` in GSM 06.10.
4. `v2=extract_h(L_v1)` is written as `v2 = L_v1` in GSM 06.10.
5. `negate(v1)` is written as `-v1` in GSM 06.10.
6. `L_negate(L_v1)` is written as `-L_v1` in GSM 06.10.
7. `L_shl(L_v1,v2)` is written as `L_v1<<v2` in GSM 06.10.
8. `L_shr(L_v1,v2)` is written as `L_v1>>v2` in GSM 06.10.
9. `L_v2=deposit_l(v1)` is written as `L_v2=v1` in GSM 06.10.
10. `div_s(v1,v2)` is written as `div(v1,v2)` in GSM 06.10.
11. `norm_l(L_v1)` is written as `norm(L_v1)` in GSM 06.10.
12. GSM 06.20 uses `shift_r(v1,v2)`, which can be implemented as `shr_r(v1,-v2)`.
13. GSM 06.20 uses `L_shift_r(L_v1,v2)`, which can be implemented as `L_shr_r(L_v1,-v2)`.
14. `div_s(v1,v2)` is written as `divide_s(v1,v2)` in GSM 06.20.
15. Operator is not part of the original ETSI library.
16. Operator is not part of the original ETSI library but was accepted in the TETRA standard.

Chapter 13

UTILITIES: UGST utilities

This module does not relate to any ITU-T Recommendation, but implements several general-purpose routines, that are needed when using other STL modules.

In the process of implementing the STL modules, it was found that the interfacing between data representations (`float` and `short`; `serial` and `parallel`) could present problems. Hence, algorithms implementation these functions have been made available in the ITU-T STL. Additionally, a scaling routine for application of gain and loss to speech samples is included.

13.1 Some definitions

Some functions in this module convert between a serial format and a parallel format. The *parallel format* is defined to be a representation in which all the bits in a computer word have an information content, as in a multi-level representation of data. Speech samples in a computer file are a typical example of a parallel representation. A *serial format* is defined as the representation of the data where each computer word refer to a single bit of information. An example would be the sequence of bits sent in a communication channel referring to an encoded digital signal. A *serial bitstream*, in the context of the ITU-T STL, refers to a multi-level representation of information bits in which each of the “hard” bits ‘0’ or ‘1’ are mapped respectively to the so-called softbits 0x007F and 0x0081, to which an error probability is associated. These softbits are stored in 16-bit right-justified words. In addition, if the bitstream is compliant to the bitstream signal representation in Annex B of ITU-T Recommendation G.192, the serial bitstream “payload” described above will be preceded by a synchronization header. A *synchronization header* is composed by a synchronization word followed by a frame length word. *Synchronization words* are words in the bitstream in the range 0x6B21 to 0x6B2F. A synchronization word equal to 0x6B20 indicates a frame loss. The *frame length word* is a two-complement word representing the number of softbits in the payload. Therefore, the frame length word does not account for the synchronization header length (which equals two, by definition). Typically (as in the EID module), encoded signals are represented using the bitstreams with a synchronization header, while error patterns are represented without a synchronization header.

13.2 Implementation

The functions implemented in this module are:

- **scale**:for level change of a float data stream;
- **sh2fl***:for conversion from **short** to **float**;
- **fl2sh**:for conversion from **float** to **short**;
- **serialize_***:for conversion from parallel to serial data representation;
- **parallelize_***:for conversion from serial to parallel data representation;

Following you find a summary of calls to these functions.

13.2.1 scale

Syntax:

```
#include "ugst-utl.h"
long scale (float *buffer, long smpno, double factor);
```

Prototype: ugst-utl.h

Description:

Gain/loss insertion algorithm that scales the input buffer data by a given factor. If the factor is greater than 1.0, it means a gain; if less than 1.0, a loss. The basic algorithm is:

$$y(k) = x(k) \cdot factor$$

Please note that:

- the scaled data is put into the same location of the original data, in order to save memory space, thus overwriting original samples;
- input data buffer is an array of **floats**;
- scaling precision is single (rather than double precision).

Variables:

<i>buffer</i>	Float data vector to be scaled.
<i>smpno</i>	Number of samples in <i>buffer</i> .
<i>factor</i>	The float scaling factor.

Return value:

Return the number of scaled samples.

13.2.2 sh2fl

Syntax:

```
#include "ugst-utl.h"
void sh2fl (long n, short *ix, float *y, long resolution, char norm);
```

Prototype: ugst-utl.h

Description:

Common conversion routine. The conversion routine expects the fixed point data to be in the range between $-32768..32767$. Conversion to float is done by taking into account only the most significant bits (i.e., input samples shall be left-justified), normalizing afterwards to the range $-1..+1$, if *norm* is 1.

In order to maintain a match with its complementary routine `fl2sh`, a set of macros have been defined for resolutions in the range of 16 to 12 bits (see below for the complementary definitions):

- *sh2fl_16bit*: conversion from 16 bit to float
- *sh2fl_15bit*: conversion from 15 bit to float
- *sh2fl_14bit*: conversion from 14 bit to float
- *sh2fl_13bit*: conversion from 13 bit to float
- *sh2fl_12bit*: conversion from 12 bit to float

Variables:

<i>n</i>	Is the number of samples in <code>ix[]</code> ;
<i>ix</i>	Is input short array pointer;
<i>y</i>	Is output float array pointer;
<i>resolution</i>	Is the resolution (number of bits) desired for the input data in the floating point representation.
<i>norm</i>	Flag for normalization: 1: normalize float data to the range $-1..+1$; 0: convert from short to float, leaving data in the range: $-32768 \gg (16 - \text{resolution}) .. 32767 \gg (16 - \text{resolution})$, where \gg is the right-shift operation.

Return value:

None.

13.2.3 sh2fl_alt

Syntax:

```
#include "ugst-utl.h"
void sh2fl_alt (long n, short *ix, float *y, short mask);
```

Prototype: `ugst-utl.h`

Description:

Common conversion routine alternative to routine `sh2fl`. This conversion routine expects the fixed-point data to be in the range $-32768..32767$. Conversion to float is done by taking into account only the most significant bits, indicated by *mask*. Conversion to float results necessarily in normalised values in the range $-1.0 \leq y < +1.0$.

Variables:

<i>n</i>	Number of samples in <code>ix[]</code> .
<i>ix</i>	Pointer to input short array.
<i>y</i>	Pointer to output float array.

mask Mask determining how many bits of the input samples are to be considered for conversion to float. Bits '1' in *mask* indicate that this bit in particular will be used in the conversion. For example, *mask* equal to 0xFFFF indicates that all 16 bits of the word are used in the conversion, while *mask* equal 0xFFFE, 0xFFFC, 0xFFF8, or 0xFFF0 will force respectively only the upper 15, 14, 13, or 12 most significant bits to be used.

Return value:

None.

13.2.4 fl2sh

Syntax:

```
#include "ugst-utl.h"
long fl2sh (long n, float *x, short *iy, double half_lsb, unsigned mask);
```

Prototype: ugst-utl.h

Description:

Common quantisation routine. The conversion routine expects the floating point data to be in the range between -1..+1, values outside this range are limited. Quantization is done by taking into account only the most significant bits. Therefore, the quantized (or converted) data are located left justified within the 16-bit word, and the results are in the range:

- -32768, ..., -1, 0, +1, ..., +32767, if quantized to 16 bit
- -32768, ..., -2, +2, ..., +32766, if quantized to 15 bit
- -32768, ..., -4, +4, ..., +32763, if quantized to 14 bit
- -32768, ..., -8, +8, ..., +32760, if quantized to 13 bit
- -32768, ..., -16, +16, ..., +32752, if quantized to 12 bit

The operation may be summarized as:

$$y_k = (x_k \pm h) \& m$$

where x_k is the float number, y_k is the quantized number, h is the value of half-LSb for the resolution desired (which is added to x_k if the latter is positive or zero, or subtracted otherwise), and m is the bit mask (to assure that the bits below the LSb are 0). The operation = is a truncation, and & is a bit-wise AND operation. The appropriate values for h are determined by:

$$h = 0.5 \cdot 2^{16-B} = 2^{15-B}$$

where B is the desired resolution in bits. As an example, if data is to be stored with 15 bits of resolution (equivalent to -16384..+16383, in right-justified notation), the rounding number h is 1.0, because the smallest number in the output buffer can be +1 or -1. The mask m , by its turn, is

$$m = 0xFFFF \ll (16 - B)$$

where \ll is the left-shift bit operation with zero-padding from the right. For the same example, m is `0xFFFE`, i.e., only bit 0 of the samples is zeroed.

To facilitate to the use of the `fl2sh`, a set of macros has been defined for quantizations in the range of 16 to 12 bits (see `ugst-utl.h`):

- `fl2sh_16bit`: conversion from float to 16 bit
- `fl2sh_15bit`: conversion from float to 15 bit
- `fl2sh_14bit`: conversion from float to 14 bit
- `fl2sh_13bit`: conversion from float to 13 bit
- `fl2sh_12bit`: conversion from float to 12 bit

In some cases truncated data is needed, what can be accomplished by setting $h = 0$. For example, at the input for A-law encoding, truncation is necessary, not rounding. On the other hand within recursive filters rounding is essential. Hence, this routine serves both cases.

Concerning the location of the fixed-point data within one 16 bit word, it is more practical to have the decimal point immediately after the sign bit (between bit 15 and 14, if the bits are ordered from 0..15). Since this is well defined, software that processes the quantized data needs no knowledge about the resolution of the data. It is not important whether the data comes from A or μ law decoding routines or from 12-bit (13, 14, 16-bit) A/D converters.

It should be noted that this routine only processes data in a normalized form ($-1.0 \leq x < +1.0$); it shall not be used if data is in the `short` range ($-32768.0 .. 32767.0$).

Variables:

<code>n</code>	Number of samples in <code>x[]</code> .
<code>x</code>	Pointer to input <code>float</code> array.
<code>iy</code>	is output <code>short</code> array pointer.
<code>half_lsb</code>	A <code>double</code> representation of half LSB for the desired resolution (quantization).
<code>mask</code>	The <code>unsigned</code> masking of the lower (right) bits.

Return value:

Returns the number of overflows that happened in the quantization process.

13.2.5 `serialize*_justified`

Syntax:

```
#include "ugst-utl.h"
long serialize_right_justified (short *par_buf, short *bit_stm, long n, long
                                resol, char sync);
long serialize_left_justified (short *par_buf, short *bit_stm, long n, long
                                resol, char sync);
```

Prototype: `ugst-utl.h`

Description:

Routines `serialize_right_justified` and `serialize_left_justified` convert a frame of n right- or left-justified samples with a resolution *resol* into a right-justified, serial soft bitstream of length $n.resol$. If the parameter *sync* is set, a serial bitstream compliant to the Annex B of ITU-T Recommendation G.192 will be generated. In this case, the length of the bitstream is increased to $(n+2).resol$.¹ It should be noted that the least significant bits of the input words are serialized first, such that the bitstream is a stream with less significant bits coming first.

The only difference between these functions is that function `serialize_right_justified` serializes right-justified parallel data and function `serialize_left_justified` serializes left-adjusted data.

It is supposed that all parallel samples have a constant number of bits, or resolution, for the whole frame. If this does not happen, the bitstream cannot be serialized by these functions. As an example, this is the case of the RPE-LTP bitstream: the 260 bits of the encoded bitstream are not divided equally among the 76 parameters of the bitstream. In cases like this, users must write their own serialization function.

Variables:

<i>par_buf</i>	Input buffer with right- or left-adjusted, parallel samples to be serialized.
<i>bit_stm</i>	Output buffer with serial bitstream. It should be noted that <i>bit_stm</i> must point to an appropriately allocated memory block, which should be a block of $n.resol$ shorts if <i>sync</i> is 0, or a block of $(n+2).resol$ shorts otherwise.
<i>n</i>	Number of words in the input buffer, i.e., the number of parallel samples/frame.
<i>resol</i>	Resolution (number of bits) of the samples in <i>par_buf</i> .
<i>sync</i>	If 1, a synchronization header is to be used (appended) at the boundaries of each frame of the bitstream. If 0, a synchronization header is not used.

Return value:

This function returns the total number of softbits in the output bitstream, including the synchronization word and frame length. If the value returned is 0, the number of converted samples is zero.

13.2.6 parallelize*_justified

Syntax:

```
#include "ugst-utl.h"

long parallelize_right_justified (short *bit_stm, short *par_buf, long bs_len,
                                long resol, char sync);

long parallelize_left_justified (short *bit_stm, short *par_buf, long bs_len,
                                long resol, char sync);
```

¹The option of adding only the synchronization word, as implemented in the STL92, is no longer available with this function since the STL96.

Prototype: ugst-utl.h

Description:

Functions `parallelize_right_justified` and `parallelize_left_justified` convert the samples in input buffer *bit_stm* from the ITU-T softbit representation to its parallel representation, given a number of bits per sample, or *resolution*. The input serial bitstream of length *bs_len* is converted into a frame with $bs_len/resol$ samples (if *sync*==0) or $(bs_len-2)/resol$ samples (if *sync*!=0), with a resolution *resol*. It should be noted that softbits in lower positions in the input buffer are supposed to represent less significant bits of the parallel word (considering bits that would compose the same parallel word). In other words, the softbits that come first are less significant than the next ones, when referring to the same parallel word (as defined by the parameter *resol*). Therefore, when generating a word from the bitstream, bits from the bitstream that comes first are converted to lower significant bits. Frames with the synchronization flag but without the frame length cause the function to exit with an error code equal to $-bs_len$.

The difference between both functions is that `parallelize_right_justified` converts the serial bitstream to a parallel data in a right-justified format, i.e., data is aligned to the right, while the routine `parallelize_left_justified` parallelizes samples with left-justification.

If the G.192 Annex B bitstream format is used (parameter *sync*==1), a synchronization header is present at frame boundaries in the input buffer. In this case, the synchronization and frame length words are not copied from the bitstream to the output buffer.

Note that all parallel samples are supposed to have a constant number of bits, or resolution, for the whole frame. This means that, by construction, the number of softbits divided by the resolution must be an integer number, or $(bs_len-2)\%resol==0$. If this does not happen, probably the serial bitstream was not generated by one of the `serialize_...()` routines, and cannot be parallelized by these functions. An example is the case of the RPE-LTP bitstream: the 260 bits of the encoded bitstream are not divided equally among the 76 parameters of the bitstream. In cases like this, users must write their own parallelization function.

If an erased frame is found, the function returns without performing any action.

Variables:

<i>bit_stm</i>	Input buffer with bitstream to be parallelized.
<i>par_buf</i>	Output buffer with right- or left-adjusted samples.
<i>bs_len</i>	Number of bits per frame (i.e., size of input buffer, which includes the synchronization header length if <i>sync</i> ==1).
<i>resol</i>	Resolution (number of bits per parallel sample) of the right- or left-adjusted samples in <i>par_buf</i> .
<i>sync</i>	If 1, a synchronization header is expected in the boundaries of each frame of input the bitstream. If 0, synchronization headers are not expected.

Return value:

On success, this function returns the number of samples of the output parallel sample buffer.

13.3 Portability and compliance

Since these tools do not refer to ITU-T recommendations, no special compliance tests are needed. As for portability, it may be checked by running the same speech file on a proven platform and on a test platform. Files processed this way should match exactly. A preferred data file would be the ramp described in the compliance test description.

The routines in this module had portability tested for VAX/VMS with VAX-C and GNU C (gcc) and for MS-DOS with a number of Borland C/C++ compilers (Turbo C v2.0, Turbo-C++ v1.0, Borland C++ v3.1). Portability was also tested in a number of Unix workstations and compilers: Sun workstation with Sun-OS and Sun-C (cc), acc, and gcc; HP workstation with HP-UX and gcc.

13.4 Example code

13.4.1 Description of the demonstration programs

Two programs are provided as demonstration programs for the UTL module, `scaldemo.c` (version 1.3) and `spdemo.c` (version 3.2).

Program `scaldemo.c` scales a 16-bit, linear PCM input file by a user-specified linear or dB gain value. Default resolution is 16 bits per sample, and rounding is used by default when converting from float to short. When resolutions different from 16 bits are used with rounding, versions 1.2 and earlier of the program might not produce the "expected" results. The program used to limit the resolution of the samples (by masking the $16 - \text{resolution}$ least significant bits) when converting from short to float. Additional rounding is applied after scaling when converting from float to short. If the desired operation is, actually, scale and then reduce the resolution with rounding, masking before the scaling operation should be disabled. In version 1.3 and later, the default behavior is **not** to apply such mask, (same as the option `-nopremask`) for backward compatible behavior, the option `-premask` should be explicitly used.

Program `spdemo.c` converts files between serial and parallel formats using a user-specified resolution and frame (or block) size. A known issue with `spdemo` version 3.2 is that the command-line option `-frame` does not work properly for parallel-to-serial conversion. In this case, the desired frame size has to be specified as parameter N in the command line.

13.4.2 The master header file for the STL demonstration programs

The module also contains the common demonstration program definition file `ugstdemo.h` (version 2.2), which is used by all STL demonstration programs. This header file contains the definition of a number of pseudo-functions and symbols that facilitate the use of a more homogeneous user interface for the different demonstration programs in the STL.

The available pseudo-functions include:

GET_PAR_* ... Pseudo-functions for printing a user prompt and reading a positional parameter from the command line. The parameters can be char (C), integers (I), long integers (L), unsigned long integers (LU), floats (F), doubles (D), and strings (S).

FIND_PAR_* .. Pseudo-functions for printing a user prompt and reading a positional parameter from the command line if it was specified by the user, or to assume a default value defined by the programmer. The parameters can be char (C), integers (I), long integers (L), floats (F), doubles (D), and strings (S).

ARGS() The pseudo-function **ARGS()** allows that the list of parameters that show up as its arguments be passed on to ANSI-C compliant compilers, or be discarded for old-vintage, K&R-style compilers that do not accept parameter list in function prototypes. This pseudo-function allows for safer function prototypes in compilers that support parameter declaration in function prototypes and avoids the need to edit function declarations (or long **#if/#else/#end** for prototype sections) for non-ANSI C compilers.

Some of the symbols defined in `ugstdemo.h` include:

- Symbols **WB**, **RB**, **WT**, **RT**, and **RWT** for file open (`fopen()`) operation. These symbols are portable across a large number of platforms and permit write-binary, read-binary, write-text, read-text, and read-write-text file mode operations.
- Symbol **MSDOS**, which is necessary for proper compilation of some of the programs under the MS-DOS environment. The symbol is defined in case MS-DOS is detected, and undefined in case MS-DOS is not detected.
- Symbol **COMPILER**, which contains a text string describing the compiler used to generate an executable.

13.4.3 Short and float conversion and scaling routines

The following C code exemplifies the use of the short and float number format interchange routines, as well as of the gain scaling routine. This program is a simplified version of the example program `scaldemo.c` provided in the STL distribution. This program reads 16-bit, 2-complement, left-justified input samples, converts them to a float representation in the range of -1..+1, applies a gain (or loss) factor to these samples, converts the scaled samples back to an integer representation (16 bit, 2's complement, left-justified) using rounding and hard-clip of the floating point numbers. The number of most significant bits to be used is also specified by the user.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "ugstdemo.h"
#include "ugst-utl.h"
```

```

#define LENGTH 5

main(argc, argv)
    int      argc;
    char     *argv[];
{
    long      i, NrSat;
    long      B, round;
    double    h;
    unsigned   m;
    short     ix[LENGTH];
    float     y[LENGTH];
    float     factor;

    GET_PAR_F(1, "_Factor: ..... ", factor);
    GET_PAR_L(2, "_Resolution: ..... ", B);
    GET_PAR_L(3, "_Round(1=yes,0=no): ... ", round);

    /* Initialize short's buffer, BUT left-adjusted! */
    for (i = 0; i < LENGTH; i++)
        ix[i] = i << (16 - B);

    /* Choose rounding number */
    h = 0.5 * (round << (16 - B));

    /* Find mask */
    m = 0xFFFF << (16 - B);

    /* Print original data */
    printf("ix before normalization\n");
    printf("=====\n");
    for (i = 0; i < LENGTH; i++)
        printf("ix[%3d]=%5d\n", i, ix[i]);

    /* Convert samples to float, normalizing */
    sh2fl(LENGTH, ix, y, B, 1);

    /* Normalizes vector */
    scale(y, LENGTH, (double) factor);

    /* Convert from float to short */
    NrSat = fl2sh(LENGTH, y, ix, h, m);

    /* Inform about overflows */
    if (NrSat != 0)
        printf("\n  Number of clippings: ..... %ld [] ", NrSat);

    /* Print new data */

```

```

printf("after normalization ... \n");
printf("=====\n");
for (i = 0; i < LENGTH; i++)
    printf("y[%3d]= %e -> ix[%3d]=%5d\n", i, y[i], i, ix[i]);

return (0);
}

```

13.4.4 Serialization and parallelization routines

The following C code implements an example of use of the serialization and parallelization routines available in the STL. Input data is generated within the program. The program takes the number of bits per sample, the justification, and whether synchronization headers should be generated. The input data is printed on the screen in its parallel representation, which is then converted to the serial format and back to the parallel format. Then, the serialized version of the data is printed on the screen, and the program ends.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "ugstdemo.h"
#include "ugst-utl.h"

#define LENGTH 5

void main(argc, argv)
    int      argc;
    char      *argv[];
{
    long      i, j, k, smpno, bitno, init;
    long      B, just;
    double     h;
    unsigned   m;
    char       c;
    short      par[LENGTH];
    short      ser[16 * LENGTH + 2];
    char       sync;
    long      (*ser_f) (); /* pointer to serialization function */
    long      (*par_f) (); /* pointer to parallelization function */

    GET_PAR_L(1, "_Resolution: ..... ", B);
    GET_PAR_L(2, "_Data is Right (1) or Left (0) justified? ... ", just);
    GET_PAR_L(3, "_Use sync header? ..... ", sync);

    /* Initialize flag "OFF" */

```

```

init = 0;
c = sync ? 1 : 0;
smpno = LENGTH;
bitno = LENGTH * B + sync ? 2 : 0;

/* Initialize data and choose pointers to appropriate functions */
if (just)
{
    /* Right-justified data */
    ser_f = serialize_right_justified;
    par_f = parallelize_right_justified;
    for (i = 0; i < LENGTH; i++)
        par[i] = i;
}
else
{
    /* Left-justified data */
    ser_f = serialize_left_justified;
    par_f = parallelize_left_justified;
    for (i = 0; i < LENGTH; i++)
        par[i] = i << (16 - B);
}

/* Print original data */
printf("\npar[] before serialization\n");
printf("=====\n");
for (i = 0; i < LENGTH; i++)
    printf("par[%3d]=%5d\n", i, par[i]);

bitno = ser_f
(par,                      /* input buffer pointer */
 ser,                      /* output buffer pointer */
 smpno,                    /* no. of samples (not bits) per frame */
 B,                        /* number of bits per sample */
 sync);                   /* whether sync header is present or not */

smpno = par_f
(ser,                      /* input buffer pointer */
 par,                      /* output buffer pointer */
 bitno,                    /* number of softbits per frame */
 B,                        /* number of bits per sample */
 sync);                   /* whether sync header is present or not */

/* Print new data */
printf("=====\n");
printf("| 0x81 represents a '1' | \n| 0x7F represents a '0' |\n");
printf("=====\n");
printf("after serialization ... \n");
printf("=====\n");
if (sync)
{

```

```
    printf("Sync word is ser[%d]= %04X", 0, ser[0]);
    printf("Frame length is ser[%d]= %04X", 1, ser[1]);
}

for (k = 2, i = 0; i < LENGTH; i++)
{
    printf("\npar[%3d]=%5d\n", i, par[i]);
    for (j = 0; j < B; j++, k++)
        printf("ser[%3d]= %04X\t", sync ? k : k - 2, ser[k]);
}
printf("\n");
}
```


Chapter 14

References

- [1] ITU-T. *Recommendation G.191, Software Tools for Speech and Audio Coding Standards*. ITU, Geneva, March 1993.
- [2] Study Group XV. Report of Working Party XV/2. Technical report, CCITT, November 1991. COM XV-R 73-E.
- [3] C. South and P. Usai. Subjective Performance of CCITT's 16 kbit/s LD-CELP Algorithm with Voice Signals. In *Globecom 92*. IEEE, 1992.
- [4] H.J. Braun, S. Feldes, and G. Schröder. Preselection for the Half-Rate GSM Standard. In *Workshop on Speech Coding for Telecommunications*, pages 90–92, September 11-13 1991.
- [5] ITU-T. *Recommendation P.48, Specification for an intermediate reference system*, volume V of *Blue Book*, pages 81–86. ITU, Geneva, 1989.
- [6] Bell Northern Research (Canada). Frequency response characteristics for low bit-rate codec testing. Technical report, UIT-T SG 12, Geneva, December 1994. Delayed Document D,38 (SG12).
- [7] ITU-T. *Recommendation P.830, Subjective performance assessment of Telephone Band and Wideband Digital Codecs*. ITU, Geneva, February 1996.
- [8] Spiros Dimolitsas, Frank Corcoran, and Channasandra Ravishankar. Correlation between headphone and telephone-handset listener opinion scores for single-stimulus voice coder performance assessments. *IEEE Signal Processing Letters*, 2(3):41–43, March 1995.
- [9] Spiros Dimolitsas, Frank Corcoran, and Channasandra Ravishankar. Dependence of opinion scores on listening sets used in degradation category rating assessments. *IEEE Transactions on Speech and Audio Processing*, 3(5):421–424, September 1995.
- [10] ITU-T. *Recommendation P.341, Characteristics of wideband terminals*. ITU, Geneva, 1994.
- [11] Denis Byrne et al. An international comparison of long-term average speech spectra. *Journal of the Acoustical Society of America*, 96(4):2108–2120, October 1994.

- [12] ITU-T. *Recommendation G.712, Performance Characteristics of PCM channels*. ITU, Geneva, 1992.
- [13] Aachen University. An Implementation of the Signal Conditioning Device. Technical Report TD91/23, ETSI/TM/TM5/TCH-HS, April 1991.
- [14] V.K. Varma. Testing speech coders for usage in wireless communications systems. In *Second IEEE Workshop on Speech Coding for Telecommunications, "Speech Coding for the Network of the Future"*, Quebec, Canada, October 13–15 1993. IEEE.
- [15] Bellcore. Proposed model for simulating radio channel burst errors. Technical report, CCITT SG XII, Geneva, October 1992. Doc.SQ-15.92(Rev.).
- [16] E.N Gilbert. Capacity of a burst-noise channel. *Bell Syst.Tech.J.*, pages 1253–1265, 1960.
- [17] D. Knuth. Seminumerical Algorithms . In *The Art of Computer Programming* . Addison-Wesley, Massachusetts, 1981.
- [18] ITU-T. *Recommendation G.192, A Common Digital Parallel Interface for Speech Standardization Activities*. ITU, Geneva, November 1995.
- [19] J. Fennick. *Quality Measures and the design of telecommunications systems* . Artech House, 1988.
- [20] ITU-T. *Recommendation G.711, Pulse code modulation (PCM) of voice frequencies*, volume Fascicle III.4 of *Blue Book*, pages 175–184. ITU, Geneva, 1989.
- [21] N.S. Jayant and P. Noll. *Digital Coding of Waveforms*. Prentice-Hall, 1984.
- [22] ITU-T. *Recommendation G.726, 40, 32, 24, 16 kbit/s adaptive differential pulse code modulation (ADPCM)*. ITU, Geneva, 1990.
- [23] M. Bonnet, O. Macchi, and M. Jaidane-Saidane. Theoretical analysis of the ADPCM CCITT algorithm. *IEEE Trans.on Communications*, 38(6):847–858, June 1990.
- [24] W.R. Daumer, X. Maitre, P. Mermelstein, and I. Tokizawa. Overview of the ADPCM coding algorithm. *Proc.Globecom*, pages 774–777, 1984.
- [25] ITU-T. Comparison of ADPCM Algorithms. *ITU-T Rec.G.726, Appendix III*, Geneva 1994.
- [26] ITU-T. Extensions of Recommendation G.726 for use with uniform quantized input and output. In *Recommendation G.726*, chapter Annex A. ITU, Geneva, 1994.
- [27] S.I. Feldman, D.M. Gay, M.W. Maimone, and N.L. Schryer. A Fortran-to-C Converter. Technical Report Computing Science 149, AT&T Bell Laboratories, August 1990.
- [28] ITU-T. *Recommendation G.727, 5-, 4-, 3- and 2-bits/sample embedded adaptive differential pulse code modulation (ADPCM)*. ITU, Geneva, December 1990.

- [29] ITU-T. *Annex A to Recommendation G.727, Extensions of Recommendation G.727 for use with uniform-quantized input and output*. ITU, Geneva, November 1994.
- [30] CCITT. *Recommendation G.722, 7 kHz audio-coding within 64 kbit/s*, volume Fascicle III.4 of *Blue Book*, pages 269–341. ITU, Geneva, 1989.
- [31] ITU-T. *Recommendation G.725, System Aspects for the Use of the 7kHz Audio Codec within 64 kbit/s*, volume Fascicle III.4 of *Blue Book*, page 11. ITU, Geneva, 1989.
- [32] Paul Mermelstein. G.722, a new CCITT coding standard for digital transmission of wideband audio signals. *IEEE Communications Magazine*, 26(1):8–15, January 1988.
- [33] Masahiro Taka et al. Overview of the 64 kbit/s (7 kHz) audio coding standard. In *Globecom 86*, pages 593–598, Houston, Texas, Dec.1–4 1986. IEEE.
- [34] G. Modena, A. Coleman, P. Usai, and P. Coverdale. Subjective Performance Evaluation of the 7 KHz Audio Coder. In *Globecom 86*, pages 599–604, Houston, Texas, Dec.1–4 1986. IEEE.
- [35] G. Le Tournier et al. Implementation of the 7 kHz Audio Codec and its Transmission Characteristics. In *Globecom 86*, pages 605–609. IEEE, 1986.
- [36] Manfred Dietrich et al. Initialization and Mode Switching of 7 kHz Audio Terminals. In *Globecom 86*, pages 610–614. IEEE, 1986.
- [37] Keith R. Harrison et al. Possible Applications for networking considerations relating to the 64 kbit (7 kHz) audio coding system. In *Globecom 86*, pages 615–622. IEEE, 1986.
- [38] Xavier Maitre. 7 kHz audio coding within 64 kbit/s. *IEEE Journal on Selected Areas in Communications*, 6(2):283–298, February 1988.
- [39] P. Kroon, R.J. Sluyter, and E.F. Deprettere. Regular-pulse excitation: a novel approach to effective and efficient multipulse coding of speech. *IEEE Trans. Acoust., Speech, Signal Processing*, ASSP-34(5):1054–1063, October 1986.
- [40] Peter Vary et al. Speech codec for the European Mobile Radio System. In *ICASSP 88*, pages 227–230. IEEE, 1988.
- [41] Ulrich Reute (Guest Editor). Special Issue on Medium Rate Speech Coding for Digital Mobile Technology. *Speech Communication*, 7(2), July 1988.
- [42] GSM-06.10. *Full Rate Speech Transcoding*. ETSI, France, October 1992. Released July 1, 1993.
- [43] ITU-T. *Recommendation P.800, Methods for the subjective determination of transmission quality*. ITU, Geneva, 1996.
- [44] H.B Law and R.A. Seymour. A reference distortion system using modulated noise. *Proc.Institution of Electrical Engineers (IEE)*, 109B:484–485, Nov 1962.

- [45] ITU-T. *Recommendation P.81, Modulated Noise Reference Unit (MNRU)* , volume V of *Blue Book*, pages 198–203. ITU, Geneva, 1989.
- [46] User's Group on Software Tools. CCITT Software Tool Library Manual. Technical report, CCITT SG XV, May 1992. COM XV-R 87-E.
- [47] ITU-T. *Recommendation P.810, Modulated Noise Reference Unit (MNRU)* . ITU, Geneva, February 1996.
- [48] S.F. Campos Neto. Characterization of the revised implementation of the Modulated Noise Reference Unit (MNRU) for the ITU-T Software Tool Library. White Contribution COM 15-182-E, ITU-T, 1993-1996.
- [49] W.H Press, B.P Flannery, S.A. Teukolky, and W.T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing* . Cambridge University Press, Cambridge, 1990.
- [50] ITU-T. *Recommendation P.56, Objective measurement of active speech level*, volume V of *Blue Book*, pages 110–120. ITU, Geneva, 1989.
- [51] ITU-T. *Handbook on Telephonometry* . ITU, Geneva, 1992. 2nd. Edition.

Appendix A

Unsupported tools

This Appendix to the ITU-T Software Tool Library (STL) Manual describes the unsupported tools provided in the ITU-T STL. These tools are provided “as is” and without any warranties or implied suitability to use. However, any feedback on problems with these tools will be welcome and accommodated as possible, as will any improvements made which can be shared and incorporated in the STL.

A.1 Source code

<i>asc2bin.c</i> :	converts decimal or hex ASCII data into short/long/float or double binary numbers. Input data must be one number per line.
<i>astrip.c</i> :	strips off a segment of a file. Can operate on block or sample-based parameters and can apply windowing to the borders of the extracted segment. Tested in Unix/MSDOS.
<i>bin2asc.c</i> :	converts short/long/float or double binary numbers into octal, decimal or hex ASCII numbers, printing one per line. For Unix/MSDOS.
<i>compfile.c</i> :	compare word-wise binary files. For VMS/Unix/MSDOS.
<i>dumpfile.c</i> :	dump a binary file. For VMS/Unix/MSDOS.
<i>chr2sh.c</i> :	convert char -oriented files to short -oriented (16-bit words) files by padding the upper byte of each word of the output file with zeros. For Unix/MSDOS.
<i>endian.c</i> :	program that verifies whether the current platform is big or little endian (i.e. high-byte first or low-byte first). For Unix/MSDOS.
<i>fdelay.c</i> :	flexible program to introduce delay into a file. Delay can be specified in value and length, or can be taken from a user-specified file. For Unix/MSDOS.
<i>g728-vt</i> :	a directory with software tools for use with the G.728 floating point verification package. Not all tools are functional; preserved here for future reference.
<i>getcrc32.c</i> :	32-bit CRC calculation function and program (depending on how it is compiled). Uses the same polynomial as ZIP. Checked for portability across a number of platforms. Makefile compiles it into an executable called <i>crc</i> . For Unix/MSDOS.
<i>measure.c</i> :	measure statistics/CRC for a bunch of files. For VMS/Unix/MSDOS.

<i>oper.c</i> :	implement arithmetic operation on two files: add, subtract, multiply or divide two files applying scaling factors (linear or dB), and adding a DC level. For Unix/MSDOS.
<i>pshar</i> :	a directory with makefiles, readme, source code and test files for a portable shell archiving/dearchiving program compatible with Unix the shar utility. Very simple and useful, in especial for MSDOS and VMS systems. See the directory for more details.
<i>sb.c</i> :	swap bytes for word-oriented files. For VMS/Unix/MSDOS.
<i>sh2chr.c</i> :	convert short -oriented (16-bit words) files to char -oriented files by ignoring the upper byte of each word of the input file. For Unix/MSDOS.
<i>sine.c</i> :	generate a sinewave file for a given speco of AC/DC/phase/ frequency/sampling frequency values. For VMS/Unix/MSDOS.
<i>sub-add.c</i> :	subtract/add files (depending on the compilation, see makefiles). For VMS/Unix/MSDOS.
<i>xencode.c</i> :	uuencode compatible with auto-break/sequencing for long files and CRC calculation for error detection. Not functional under MSDOS 6.22.
<i>xdecode.c</i> :	uudecode compatible with auto-break/sequencing for long files and CRC calculation for error detection. Not functional under MSDOS 6.22.

A.2 Scripts

<i>rm.bat</i>	“fake” deletion utility that tries to emulate the basic functionality of the Unix command rm , that deletes multiple files specified in the command line. Should be put in the path, unless a version of rm is already available.
<i>swapover.bat</i>	MSDOS batch script for byte-swapping multiple files. Uses <i>sb.c</i> .
<i>swapover.sh</i>	Unix script for byte-swapping multiple files. Uses <i>sb.c</i> .

A.3 Makefiles

<i>makefile.tcc</i>	for Borland [bt]cc C/C++ compiler
<i>makefile.djc</i>	for MSDOS djc port of gcc
<i>makefile.unx</i>	for Unix make
<i>makefile.cl</i>	for MS Visual C command-line compiler

A.4 Test files

<i>tstunsup.zip</i>	zip archive with test files for testing some of the unsupported tools:
---------------------------	--

```
cf:
  3200    cfctest1.dat
  3200    cfctest2.dat
  3200    cfctest3.dat
  186     delay-15.ref
  186     delay-a.ref
  214     delay-u.ref
  186     delaydft.ref
  200     delayfil.ref
```

```
sb:
  100     bigend.src
  100     litend.src
```

```
xencode and xdecode:
  9182    voice.ori
  8705    voice.uue
  2093    printme.eps
  3795    printme.uue
  5368    voice01.uue
```

It is necessary to have unzip/pkunzip/Winzip installed for extraction.

Appendix B

Future work

The following items have been identified as future action items by the UGST in the close future. Contributions are welcome, even though the proposed algorithm implementations may not be in full compliance with the software tool guidelines:

<i>G.728</i>	LDCELP coding at 16 kbit/s, possibly at other bitrates as well
<i>P.50, P.59</i>	Reference implementations of the Artificial Speech and Artificial Conversational Speech.
<i>Channel models</i>	Transmission channel models to be incorporated in the EID module, e.g. satellite, cellular, and IP transmission channels.
<i>Reference systems</i>	Alternatives to the MNRU, e.g. T-reference, S-Reference, PMNRU, PMNRU).
<i>G.726 & G.727 Annex A</i>	linear-interfaced G.726 and G.727.
<i>G.711 PLC</i>	Packet loss concealment algorithm for G.711 (G.711 Appendix I).
<i>Processing framework</i>	A processing framework tool for the implementation of host laboratory functions.
